

Understanding Assembly Language

(リバースエンジニアリング入門)

どうしてタイトルが2つあるの？参照: [on page ix](#)

Dennis Yurichev
[my emails](#)



©2013-2022, Dennis Yurichev.

この作品は、クリエイティブ・コモンズの表示 - 継承 4.0 国際 (CC BY-SA 4.0) の下でライセンスされています。このライセンスのコピーを表示するには、
<https://creativecommons.org/licenses/by-sa/4.0/>を訪れてください。

Text version (2023 年 10 月 23 日).

この本の最新版 (そしてロシア版) は以下で見ることができます:
<https://beginners.re/>.

翻訳者求む！

この作品を英語とロシア語以外の言語に翻訳するのを手伝ってください。どのように翻訳されたテキストを私に送っても (どれほど短くても)、私はLaTeXのソースコードに入れます。

[ここを読んでください.](#)

スピードは重要ではありません。なぜなら、これはオープンソースプロジェクトなのですから。あなたの名前はプロジェクト寄稿者として言及されます。韓国語、中国語、ペルシャ語は出版社によって予約されています。英語とロシア語のバージョンは自分でやっていますが、私の英語はまだひどいので、文法などに関するメモにはとても感謝しています。私のロシア語にも欠陥があるので、ロシア語のテキストについての注釈にも感謝しています！

だから私に連絡するのをためらうことはありません: [my emails](#)

簡略版

1	コードパターン	1
2	Japanese text placeholder	545
3		554
4	Java	555
5		556
6	ツール	557
7	その他	562
8	読むべき本/ブログ	563
	Afterword	567
	頭字語	569
	用語	572
	索引	574

目次

1	コードパターン	1
1.1	方法	1
1.2	基本的な事柄	2
1.2.1	簡単なCPU入門	2
1.2.2	数値システム	3
1.2.3	1つの基数から別の基数への変換	4
1.3	空関数	7
1.3.1	x86	7
1.3.2	ARM	7
1.3.3	MIPS	8
1.3.4	実際の空関数	8
1.4	戻り値	9
1.4.1	x86	9
1.4.2	ARM	10
1.4.3	MIPS	10
1.5	ハローワールド!	11
1.5.1	x86	11
1.5.2	x86-64	18
1.5.3	ARM	23
1.5.4	MIPS	32
1.5.5	結論	38
1.5.6	練習問題	38
1.6	関数のプロローグとエピローグ	38
1.6.1	再帰	38
1.7	スタック	39
1.7.1	スタックはなぜ後方に進むのか	39
1.7.2	スタックは何に使用されるか	40
1.7.3	典型的なスタックレイアウト	48
1.7.4	スタックのノイズ	49
1.7.5	練習問題	54
1.8	printf() 引数を取って	54
1.8.1	x86	54
1.8.2	ARM	67
1.8.3	MIPS	74
1.8.4	結論	82
1.8.5	ところで	83
1.9	scanf()	84
1.9.1	Simple example	84

1.9.2 一般的な間違い	96
1.9.3 グローバル変数	96
1.9.4 scanf()	108
1.9.5 練習問題	121
1.10 渡された引数にアクセスする	122
1.10.1 x86	122
1.10.2 x64	125
1.10.3 ARM	129
1.10.4 MIPS	133
1.11 戻り値を返すことの詳細	134
1.11.1 void を返す関数の結果を試してみる	134
1.11.2 関数の戻り値を使わないとどうなる？	136
1.11.3 構造体を返す	136
1.12 ポインタ	138
1.12.1 戻り値	138
1.12.2 入力値の入れ替え	148
1.13 GOTO演算子	149
1.13.1 デッドコード	152
1.13.2 練習問題	153
1.14 条件付きジャンプ	153
1.14.1 シンプルな例	153
1.14.2 絶対値の計算	174
1.14.3 三項条件演算子	176
1.14.4 最小値と最大値の取得	181
1.14.5 結論	187
1.14.6 練習問題	188
1.15 switch()/case/default	189
1.15.1 小さな数のcase	189
1.15.2 A lot of cases	204
1.15.3 あるブロックに複数の case 文があるとき	218
1.15.4 フォールスルー	223
1.15.5 練習問題	225
1.16 ループ	225
1.16.1 単純な例	225
1.16.2 メモリブロックコピールーチン	239
1.16.3 条件チェック	242
1.16.4 結論	243
1.16.5 練習問題	245
1.17 文字列に関する加筆	246
1.17.1 strlen()	246
1.17.2 文字列境界	259
1.18 算術命令を他の命令に置換する	260
1.18.1 乗算	260
1.18.2 除算	266
1.18.3 練習問題	267
1.19 フローティングポイントユニット	268
1.19.1 IEEE 754	268
1.19.2 x86	268
1.19.3 ARM, MIPS, x86/x64 SIMD	268

	v
1.19.4 C/C++	268
1.19.5 簡単な例	269
1.19.6	280
1.19.7 比較の例	283
1.19.8 いくつかの定数	321
1.19.9 コピー	321
1.19.10 スタック、計算機と逆ポーランド記法	322
1.19.11 80ビット?	322
1.19.12 x64	322
1.19.13 練習問題	322
1.20 配列	322
1.20.1 単純な例	322
1.20.2 バッファオーバーフロー	332
1.20.3 バッファオーバーフロー保護手法	340
1.20.4 配列についてもう少し	345
1.20.5 文字列へのポインタの配列	346
1.20.6 多次元配列	355
1.20.7 2次元配列としての文字列のパック	364
1.20.8 結論	368
1.20.9 練習問題	369
1.21 特定のビットを操作する	369
1.21.1 特定のビットチェック	369
1.21.2 特定ビットの設定とクリア	374
1.21.3 シフト	384
1.21.4 特定ビットのセットやクリア: FPU ¹ の例	384
1.21.5 Counting bits set to 1	390
1.21.6 結論	408
1.21.7 練習問題	411
1.22 線形合同生成器	411
1.22.1 x86	412
1.22.2 x64	413
1.22.3 32ビットARM	414
1.22.4 MIPS	415
1.22.5 スレッドセーフ版の例	418
1.23 構造体	418
1.23.1 MSVC: SYSTEMTIME example	418
1.23.2 malloc() を使って構造体のための領域を割り当てよう	422
1.23.3 UNIX: struct tm	425
1.23.4 フィールドを構造体にパッキングする	437
1.23.5 構造体の入れ子	446
1.23.6 構造体のビットフィールド	449
1.23.7 練習問題	458
1.24 共用体	458
1.24.1 擬似乱数生成器の例	458
1.24.2 計算機イプシロンを計算する	462
1.24.3 FSCALE replacement	465
1.24.4 高速平方根計算	466

¹Floating-Point Unit

1.25 関数へのポインタ	467
1.25.1 MSVC	468
1.25.2 GCC	475
1.25.3 関数へのポインタの危なさ	480
1.26 32ビット環境での64ビット値	481
1.26.1 64ビットの値を返す	481
1.26.2 Arguments passing, addition, subtraction	482
1.26.3 乗算、除算	486
1.26.4 右シフト	491
1.26.5 32ビット値から64ビット値への変換	492
1.27 SIMD	493
1.27.1 ベクトル化	495
1.27.2 SIMD strlen() 実装	507
1.28 64ビット	512
1.28.1 x86-64	512
1.28.2 ARM	521
1.28.3 Float point numbers	521
1.28.4 64-bit architecture criticism	521
1.29 SIMDを使用した浮動小数点数の取り扱い	521
1.29.1 単純な例	522
1.29.2 引数を介して浮動小数点数を渡す	530
1.29.3 Comparison example	531
1.29.4 計算機イプシロンを計算する: x64 と SIMD	534
1.29.5 疑似乱数値の生成例を再訪	535
1.29.6 概要	535
1.30 ARM固有の詳細	536
1.30.1 番号の前の番号記号 (#)	536
1.30.2 アドレッシングモード	536
1.30.3 レジスタへの定数のロード	537
1.30.4 ARM64での再配置	540
1.31 MIPS特有の詳細	542
1.31.1 32ビット定数をレジスタにロードする	542
1.31.2 MIPSについてさらに読む	544
2 Japanese text placeholder	545
2.1 Integral datatypes	545
2.1.1 Bit	545
2.1.2 Nibble AKA nybble	546
2.1.3 Byte	547
2.1.4 Wide char	548
2.1.5 Signed integer vs unsigned	548
2.1.6 Word	548
2.1.7 Address register	550
2.1.8 Numbers	550
3	554
4 Java	555
4.1 Java	555

4.1.1	555
4.1.2	555
4.1.3	555
5	556
5.1 Linux	556
5.2 Windows NT	556
5.2.1 Windows SEH	556
6 ツール	557
6.1 バイナリ解析	557
6.1.1 ディスアセンブラ	558
6.1.2 デコンパイラ	558
6.1.3 パッチの比較/diffing	558
6.2 ライブ解析	558
6.2.1 デバッガ	558
6.2.2 ライブラリコールトレース	559
6.2.3 システムコールトレース	559
6.2.4 ネットワーク傍受	560
6.2.5 Sysinternals	560
6.2.6 Valgrind	560
6.2.7 エミュレータ	560
6.3 他のツール	561
6.3.1 電卓	561
6.4 何か足りないものは？	561
6.5	561
6.6	561
7 その他	562
8 読むべき本/ブログ	563
8.1 本と他の資料	563
8.1.1 リバースエンジニアリング	563
8.1.2 Windows	563
8.1.3 C/C++	564
8.1.4 x86 / x86-64	564
8.1.5 ARM	564
8.1.6 アセンブリ言語	565
8.1.7 Java	565
8.1.8 UNIX	565
8.1.9 プログラミング一般	565
8.1.10 暗号学	565
Afterword	567
8.2 Questions?	567

頭字語	569
用語	572
索引	574

はじめに

どうしてタイトルが2つあるの？

2014年～2018年に “Reverse Engineering for Beginners” という名前を付けていましたが、読者層が狭すぎるといつも思っていました。

情報セキュリティの人々は “リバースエンジニアリング” について知っていますが、私はめったに “アセンブラ” という言葉を聞きませんでした。

同様に、“リバースエンジニアリング” という用語は、一般的なプログラマの読者にとってはやや暗黙の言葉ですが、“アセンブラ” については知っています。

2018年7月、実験として “初心者のためのアセンブリ言語” のタイトルを変更し、Hacker Newsのウェブサイトへのリンク²を掲載しました。この本は一般によく受け入れられました。

そんなわけでさすがにそのままにして、タイトルを2つにしてあります。

しかし、2番目のタイトルを “アセンブリ言語を理解する” に変更しました。これは、すでに誰かが “初心者のためのアセンブリ言語” という本を既にも書いているからです。また、“初心者のため” という言葉は～1000ページ以上ある本だと皮肉に聞こえます。

2つの書籍は、タイトルとファイル名 (UAL-XX.pdfに対してRE4B-XX.pdf)、URLと最初の数ページのみ異なります。

リバースエンジニアリングについて

「[reverse engineering](#)」にはよく知られた意味がいくつかあります。

- 1) ソフトウェアのリバースエンジニアリング、コンパイルされたプログラムの研究
- 2) 3D構造のスキャンと、それらを複製するために必要なその後のデジタル操作
- 3) [DBMS](#)³ 構造の再構成

本書は最初の意味についての本です。

前提条件

Cプログラミング言語 ([PL](#)⁴) の基礎知識。推奨図書: [8.1.3 on page 564](#)

練習問題やタスク

...<http://challenges.re> にあります。

賛辞

<https://beginners.re/#praise>.

²<https://news.ycombinator.com/item?id=17549050>

³Database Management Systems

⁴プログラミング言語

謝辞

忍耐強く質問に答えてくれた方々：SkullC0DEr

ミスや不正確な記述を指摘してくれた方々：Alexander Lysenko, Federico Ramondino, Mark Wilson, Razikhova Meiramgul Kayratovna, Anatoly Prokofiev, Kostya Begunets, Valentin “netch” Nechayev, Aleksandr Plakhov, Artem Metla, Alexander Yastrebov, Vlad Golovkin⁵, Evgeny Proshin, Alexander Myasnikov, Alexey Tretiakov, Oleg Peskov, Pavel Shakhov, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon⁶, Ben L., Etienne Khan, Norbert Szetei⁷, Marc Remy, Michael Hansen, Derk Barten, The Renaissance⁸, Hugo Chan, Emil Mursalimov, Tanner Hoke, Tan90909090@GitHub, Ole Petter Orhagen, Sourav Punoriyar, Vitor Oliveira, Alexis Ehret, Maxim Shlochiski, Greg Paton, Pierrick Lebourgeois, Abdullah Alomair, Bobby Battista, Ashod Nakashian.

他に手助けしてくれた方々：Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha “tiphareth” Verbitsky.

簡体字中国語への翻訳：Antiy Labs(antiy.cn)、Archer

韓国語への翻訳：Byungho Min

オランダ語への翻訳：Cedric Sambre (AKA Midas)

スペイン語への翻訳：Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames, Emiliano Estevearena

ポルトガル語への翻訳：Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe, Primo David Santini

イタリア語への翻訳：Federico Ramondino⁹, Paolo Stivanin¹⁰, twyK, Fabrizio Bertone, Matteo Sticco, Marco Negro¹¹, bluepulsar

フランス語への翻訳：Florent Besnard¹², Marc Remy¹³, Baudouin Landais, Téo Dacquet¹⁴, BlueSkeye@GitHub¹⁵

ドイツ語への翻訳：Dennis Siekmeier¹⁶, Julius Angres¹⁷, Dirk Loser¹⁸, Clemens Tamme, Philipp Schweinzer, Tobias Deiminger

ポーランド語への翻訳：Kateryna Rozanova, Aleksander Mistewicz, Wiktoria Lewicka, Marcin Sokołowski

⁵goto-vlad@github

⁶<https://github.com/pixjuan>

⁷<https://github.com/73696e65>

⁸<https://github.com/TheRenaissance>

⁹<https://github.com/pinkrab>

¹⁰<https://github.com/paolostivanin>

¹¹<https://github.com/Internaut401>

¹²<https://github.com/besnardf>

¹³<https://github.com/mremy>

¹⁴<https://github.com/T30rix>

¹⁵<https://github.com/BlueSkeye>

¹⁶<https://github.com/DSiekmeier>

¹⁷<https://github.com/JAngres>

¹⁸<https://github.com/PolymathMonkey>

日本語への翻訳 : shmz@github¹⁹, 4ryuJP@github²⁰.

校正者 : Vladimir Botov, Andrei Brazhuk, Mark “Logxen” Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev²¹ は大量の校正とミスを訂正してくれました。

github.comでフォークして指摘や訂正してくれたすべての方々に感謝します。

LaTeX パッケージをたくさん使っています : パッケージの作者にも合わせて感謝します

ドナー

本の大半を執筆中にサポートしてくれた方々 :

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar (\$50), James Truscott (\$4.5), Luis Rocha (\$63), Joris van de Vis (\$127), Richard S Shultz (\$20), Jang Minchang (\$20), Shade Atlas (5 AUD), Yao Xiao (\$10), Pawel Szczur (40 CHF), Justin Simms (\$20), Shawn the Rock (\$27), Ki Chan Ahn (\$50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao (\$50), Philippe Teuwen (\$4), Martin Haeberli (\$10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai (\$15), Bayna AlZaabi (\$75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop (\$50), Nicolas Werner (€12), Jeremy Brown (\$100), Alexandre Borges (\$25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent (\$30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe (\$14), Maxim Dyakonov (\$3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov (\$100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei (\$40), Zovsky (€10), Yu Dai (\$10), Anonymous (\$15), Vladislav Chelnokov (\$25), Nenad Noveljic (\$50), Ryan Smith (\$25), Andreas Schommer (€5), Nikolay Gavrilov (\$300), Ernesto Bonev Reynoso (\$30).

ドナーの方すべてに感謝します !

mini-FAQ

Q : この本を読むための前提条件は何ですか ?

A : C/C++の基本的な理解があるのが望ましいです。

Q : x86/x64/ARMとMIPSを本当にすぐに学ばべきでしょうか? それはあまりにも大変ではないですか ?

A : 初心者は、ARMとMIPSの部分をスキップまたはスキミングしながら、x86/x64だけを読んでもいいです。

Q : ロシア語または英語のハードカバー/ペーパーブックを購入できますか ?

A : 残念ながら、いいえ。これまでにロシア語版や英語版を出版することに興味を持った出版社はいませんでした。その間に、お気に入りのコピーショップに印刷してバインドス

¹⁹<https://github.com/shmz>

²⁰<https://github.com/4ryuJP>

²¹<https://vasil.ludost.net/>

るよう依頼することができます。https://yurichev.com/news/20200222_printed_RE4B/.

Q : epubまたはmobiのバージョンはありますか？

A : いいえ。本はTeX / LaTeX固有のハッキングに強く依存しているので、HTML (epub / mobiはHTMLの集合です) への変換は簡単ではありません。

Q : 最近、アセンブリ言語を学ばなければならないのはなぜですか？

A : あなたがOS²²開発者でない限り、アセンブラでコード化する必要はないでしょう。最新のコンパイラ (2010s) は、人間よりも最適化を実行する方がはるかに優れています²³

また、最新のCPUは非常に複雑なデバイスであり、アセンブリの知識は内部を理解するのに役立つものではありません。

それは、少なくとも2つの領域があり、アセンブリの理解を深めることが役立つことがあります。まず、セキュリティ/マルウェアの研究に役立つことです。また、デバッグ中にコンパイルされたコードをよりよく理解するための良い方法です。したがって、この本は、アセンブリ言語を記述するのではなく、アセンブリ言語を理解したい人のために用意されています。そのため、コンパイラ出力の例が多数含まれています。

Q : PDF文書内のハイパーリンクをクリックしましたが、どのように戻ってきますか？

A : Adobe Acrobat ReaderでAlt+左矢印をクリックします。Evinceで “ < ” ボタンをクリックしてください。

Q : この本を印刷して教えてもいいですか？

A : もちろん！だからこの本はクリエイティブコモンズライセンス (CC BY-SA 4.0) に基づいてライセンスされています。

Q : なぜこの本は無料ですか？あなたは素晴らしい仕事をしてきました。これは他の多くの自由なものと同様に疑わしいものです。

A : 私自身の経験では、技術文献の著者は主に自己宣伝の目的で書いています。そのような仕事からまともな金を稼ぐことはできません。

Q : リバースエンジニアリングではどのように仕事をしていますか？

A : redditには時々現れる採用スレッドがあり、RE²⁴に専念しています。そこを見てみてください。

多少関連する採用スレッドは、netsecサブディレクトリにあります。

Q : 質問があります...

A : 私にメールしてください ([my emails](#))

韓国語版について

2015年1月、韓国のAcorn出版社 (www.acornpub.co.kr) は、この書籍を2014年8月に韓国語に翻訳して出版する際に膨大な作業をしました。

²²オペレーティングシステム

²³このトピックに関する非常に良いテキスト : [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

²⁴reddit.com/r/ReverseEngineering/

現在、[彼らのウェブサイト](#)で利用可能です。

翻訳者はByungho Min ([twitter/tais9](#)) です。カバーアートは、作家の友人Andy Nechaevsky([facebook/andydi](#))によって行われました。Acornには、韓国語翻訳の著作権もあります。

そして、あなたが韓国語で本棚にリアルな本がほしくて、この作品をサポートしたいなら、現在購入可能です。

ペルシア/ファルシ語版について

2016年にこの本はMohsen Mostafa Jokar (Radadeのマニュアルを翻訳していて、イランのコミュニティにも知られています) によって翻訳されました。出版社のウェブサイト (Pendare Pars) で入手できます。

ここに40ページの抜粋へのリンクがあります : <https://beginners.re/farsi.pdf>

イラン国立図書館登録情報 : <http://opac.nlai.ir/opac-prod/bibliographic/4473995>

中国語版について

2017年4月、中国語への翻訳は中国のPTPressによって完了しました。中国語の著作権者でもあります。

中国語版はこちらから注文できます : <http://www.epubit.com.cn/book/details/4174>

翻訳の背後にある部分的なレビューと履歴は、ここにありますが : <http://www.cptoday.cn/news/detail/3155>

主な翻訳者はArcherです。彼は非常に細心の注意を払っており、この本のような文献では非常に重要な既知の間違いやバグのほとんどを報告していました。私は彼のサービスを他の著者に推薦します！

Anti Labsのスタッフも翻訳を手伝ってくれました。[ここ](#)に彼らが書いた序文があります。

第 1 章

コードパターン

第1.1節方法

この本の著者はC言語、その後Cppを学び始めたとき、小さなコードを書いてコンパイルし、アセンブリ言語の出力を見ていました。これにより、彼が書いたコードで何が起きているのかを理解することが非常に容易になりました。¹ 彼はこれを何度もやって、Cppコードとコンパイラが作り出したものとの関係が彼の心の中に深く刻まれていたことを知っています。今では、Cコードの外観と機能の概要を即座に想像するのは簡単です。おそらく、このテクニックは他の人に役立つかもしれません。

なお、PCにインストールせずに、さまざまなコンパイラを使ってPCと同じことができる素晴らしいWebサイトがあります。あなたもそれを使うことができます：<https://godbolt.org/>

練習問題

この本の作者がアセンブリ言語を学んだとき、彼はしばしば小さなC関数をコンパイルしてから、アセンブリを徐々に書き直してコードを可能な限り短くしようとしました。効率性の点で最新のコンパイラと競争するのは難しいため、現実のシナリオではこれはおそらく価値がありません。しかし、それはアセンブリのより良い理解を得るための非常に良い方法です。したがって、この本の中からアセンブリコードを取り出して短くしてみてください。しかし、あなたが書いたものをテストすることを忘れないでください。

最適化レベルとデバッグ情報

ソースコードはさまざまな最適化レベルを持つ異なるコンパイラによってコンパイルできます。典型的なコンパイラにはこのようなレベルが約3つあります。レベル0は最適化が完全に無効になっていることを意味します。最適化は、コードサイズやコードの速度に合わせることもできます。最適化されていないコンパイラはより高速でより理解しやすいコードを生成しますが、最適化コンパイラは遅くなり、実行速度の速いコードを生成しようと

¹ 実際、彼はコードの特定のビットが何をしているのか理解できないときもこれを実行します

します（コンパクトである必要はありません）。最適化レベルに加えて、コンパイラは結果ファイルにいくつかのデバッグ情報を含めて、デバッグしやすいコードを生成することができます。「デバッグ」コードの重要な機能の1つは、ソースコードの各行とそれぞれのマシンコードアドレスとの間にリンクを含む可能性があることです。一方、コンパイラを最適化すると、ソースコードの行全体が最適化され、結果のマシンコードにも存在しない出力が生成される傾向があります。リバースエンジニアはいずれかのバージョンに遭遇する可能性があります。なぜなら、一部の開発者はコンパイラの最適化フラグをオンにし、他の開発者はそうしないからです。このため、可能であれば、本書に記載されているコードのデバッグ版とリリース版の両方の例を取り上げようとしています。

最も短い（または最も単純な）コードスニペットを得るために、時にはかなり古いコンパイラがこの本で使われています。

第1.2節基本的な事柄

第1.2.1節簡単なCPU入門

CPU²は、プログラムが構成するマシンコードを実行するデバイスです。

短い用語集

命令：プリミティブ**CPU**コマンド。最も単純な例としては、レジスタ間のデータの移動、メモリの操作、プリミティブ算術演算などがあります。一般に、各CPUには独自の命令セットアーキテクチャ（ISA）があり、

マシンコード：CPUが直接処理するコード。各命令は、通常、数バイトで符号化されます。

アセンブリ言語：ニーモニックコードと、マクロのようないくつかの拡張機能は、プログラマーの人生をより簡単にするためのものです。

CPUレジスタ：各CPUには汎用レジスタ（GPR）の固定セットがあります。x86では約8、x86-64では約16、ARMでは約16です。レジスタを理解する最も簡単な方法は、それを型なしの一時変数と考えることです。高水準のPLで作業していて、8つの32ビット（または64ビット）変数しか使用できないとしたらどうでしょうか？しかし、これらを使って多くのことを行うことができます！

機械コードとPLの違いが必要な理由は不思議です。答えは、人間とCPUが似ていないという事実にあります。人間がCpp、Java、Pythonなどの高レベルのPLを使う方がはるかに簡単ですが、CPUがはるかに低いレベルを使用の方が簡単です。抽象化のおそらく、高レベルのPLコードを実行できるCPUを発明することは可能かもしれませんが、今日われわれが知っているCPUの何倍も複雑なものになるでしょう。同様の方法で、人間がアセンブリ言語で書くことは非常に不便です。なぜなら、それは低レベルであり、厄介な間違いを大量に作成することなく書き込むことが難しいからです。上位PLコードをアセンブリに変換するプログラムをコンパイラと呼びます。³

²Central Processing Unit

³旧式のロシア文学でも、翻訳者という用語が使われています。

異なる ISA⁴s について2、3

x86 ISAは常に可変長命令を持っていたので、64ビット時代になるとx64拡張はISAに非常に大きな影響を与えませんでした。実際、x86 ISAには、16ビットの8086 CPUに最初に登場した命令がまだ多く含まれていますが、今日のCPUではまだ見つかっています。ARMは一定の長さの命令を念頭に置いて設計されたRISC⁵ CPUであり、過去にいくつかの利点がありました。当初、すべてのARM命令は4バイトでエンコードされていました⁶。これは現在、「ARMモード」と呼ばれています。それから、彼らは最初に想像したほど儉約的ではないことに気付きました。実際のアプリケーションで最も一般的なCPU命令（MOV/PUSH/CALL/jccなど）は、より少ない情報を使用してエンコードできます。したがって、Thumbと呼ばれる別のISAを追加しました。そこでは、各命令はわずか2バイトでエンコードされていました。これを「Thumbモード」と呼びます。ただし、すべてのARM命令が2バイトでエンコードできるわけではないため、Thumb命令セットは多少制限されています。ARMモードとThumbモード用にコンパイルされたコードは、1つのプログラム内で共存することに注意してください。ARMの作成者は、Thumbを拡張して、ARMv7に登場したThumb-2を生み出すことができると考えました。Thumb-2はまだ2バイトの命令を使用しますが、4バイトのサイズを持ついくつかの新しい命令があります。Thumb-2はARMとThumbが混在しているという誤解が一般的です。これは間違っています。むしろThumb-2はすべてのプロセッサ機能を完全にサポートするように拡張されており、ARMモードと競合する可能性があります。これは明らかに達成された目標で、idevicesの大部分のアプリケーションはThumb-2命令セット用にコンパイルされています。（確かに、これは主にXcodeがデフォルトで行うためです）。後で64ビットARMができました。このISAには4バイトの命令があり、Thumbモードを追加する必要はありませんでした。しかし、64ビットの要件がISAに影響を与え、ARMモード、Thumbモード（Thumb-2を含む）、ARM64という3つのARM命令セットを持つようになりました。これらのISAは部分的に交差するが、同じISAであると言える。したがって、この本では3つのARM ISAすべてにコードの断片を追加しようとしています。ところで、MIPS、PowerPC、Alpha AXPなど固定長の32ビット命令を持つ他の多くのRISC ISAがあります。

第1.2.2節数値システム

Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.

Alan A. A. Donovan, Brian W. Kernighan —
The Go Programming Language

おそらくほとんどの人に10本の指があるので、人間は10進数字システムに慣れてきました。それにもかかわらず、数「10」は科学と数学では重要な意味を持ちません。デジタル電子機器の自然数システムはバイナリです。0は電線に電流が流れていないことを表し、1は存在を表します。バイナリで10は10進数で2、バイナリで100は小数点で4などです。

数値システムが10桁の場合、基数は10です。2進数字システムの基数は2です。

⁴Instruction Set Architecture

⁵Reduced Instruction Set Computing

⁶固定長命令は、労力を要することなく次（または前）の命令アドレスを計算できるため、便利です。この機能については、switch () オペレータセクションで説明します。

思い出すべき重要なこと：

- 1) 数字は数字であり、桁は書記体系からの言葉であり、通常は1文字
- 2) 数値の値は別の基数に変換されても変更されません。その値に対する書き込み表記だけが変更されています（したがって、RAM⁷で表現する方法）。

第1.2.3節1つの基数から別の基数への変換

位置表記はほぼすべての数値システムで使用されます。これは、数字が大きな数字の中に置かれている場所に対する相対的な重みを持つことを意味します。2が右端に置かれている場合は2ですが、右端の前に1桁置かれている場合は20です。

1234は何を表しますか？

$$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234 \text{ or } 1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$$

バイナリの数字は同じですが、ベースは10ではなく2です。0b101011は何を表していますか？

$$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43 \text{ or } 32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$$

ローマ数字のような非定位表記のようなものがあります。⁸ おそらく人類は紙で基本的な操作（加算、乗算など）を手作業で行う方が簡単であるため、位置表記法に切り替えました。

二進数は、学校で教えられたのと同じように追加、減算などが可能ですが、2桁しか利用できません。

2進数は、ソースコードとダンプで表現されているときにはかさばります。したがって、16進数表記が役立ちます。16進の基数は、0..9の数字と6つのラテン文字A..Fを使用します。各16進数字は4ビットまたは4バイナリの数字を取るので、バイナリの数字から16進数に変換したり、手動でさえ戻したりするのは非常に簡単です。

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14

⁷Random-Access Memory

⁸数値システムの進化については、195-213を参照してください。

F	1111	15
---	------	----

特定のインスタンスでどの基数が使用されているかをどのように知ることができますか？

小数点は通常1234のように書かれます。アセンブラの中には小数点以下の基数に識別子を付けることができますが、数字には接尾辞d (1234d) が付きます。

2進数には、接頭辞"0b" が付いていることがあります：0b100110111 ([GCC⁹](#)にはこのための非標準言語拡張があります¹⁰)

もう1つの方法もあります。たとえば、"b" 接尾辞を使用します (例：100110111b)。この本では、バイナリ番号のために本の中で一貫して"0b" という接頭辞を使用しようとしています。

16進数の先頭には、Cppや他のPL：0x1234ABCDの接頭辞「0x」が付加されています。あるいは、"h" 接尾辞1234ABCDhが与えられます。これはアセンブラとデバッガでそれらを表現する一般的な方法です。この規則では、数字がLatin (A..F) 桁で始まる場合、先頭に0が追加されます (0ABCDh)。ABCDのような\$接頭辞を使って8ビットの家庭用コンピュータ時代に普及した大会もありました。この本は16進数のために本の中に"0x" というプレフィックスを付けようとしています。

数字を精神的に変換することを学ぶべきでしょうか？1桁の16進数の表を簡単に記憶できます。大きな数字については、おそらく自分自身を苦しめる価値はありません。

おそらく最も目に見える16進数はURLにあります。これは非ラテン文字がコード化される方法です。たとえば<https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9>は、「naïveté」という単語に関するWiktionaryの記事のURL¹¹です。

8進数

コンピュータプログラミングの過去に多用された別の数字システムは8進数である。8進数では8桁 (0..7) であり、それぞれが3ビットにマッピングされるので、数値を前後に変換するのは簡単です。ほぼすべての場所で16進法に取って代わられていますが、驚くべきことに、多くの人が頻繁に使う*NIXユーティリティがあります。これは引数として8進数をとります (chmod)。

多くの*NIXユーザーが知っているように、chmod 引数は3桁の数字にすることができます。最初の桁はファイル所有者の権利 (読み込み、書き込み、実行) を表し、2番目はファイルが属するグループの権利で、3番目は他の人の権利です。chmod がとる各数字はバイナリ形式で表すことができます：

decimal	binary	meaning
7	111	rwX
6	110	rw-
5	101	r-X
4	100	r--
3	011	-wX

⁹GNU Compiler Collection

¹⁰<https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html>

¹¹Uniform Resource Locator

2	010	-w-
1	001	--x
0	000	---

したがって、各ビットはフラグread / write / executeにマップされます。

ここで `chmod` の重要性は、引数の整数全体を8進数で表現できることです。`chmod 644 file` を実行すると、所有者の読み取り/書き込み権限、グループの読み取り権限、他のユーザーの読み取り権限が再度設定されます。8進数644を2進数に変換すると、110100100、または3ビットのグループで 110 100 100 になります。

各トリプレットは所有者/グループ/その他のパーミッションを記述しています。最初は `rw-`、2番目は `r--`、3番目は `r--` です。

8進数システムは、PDP-8のような古いコンピュータでも人気がありました。なぜなら、そこには12,24、または36ビットが存在する可能性があり、これらの数値はすべて3で割り切れるからです。今日、普及しているすべてのコンピュータは16,32,64ビットのワード/アドレスサイズを使用しており、これらの数値はすべて4で割り切れるため、16進数のシステムはより自然です。

すべての標準C++コンパイラでサポートされています。これは混乱の原因となることがあります。なぜなら、8進数はゼロの前に付加されています（0377は255など）。時には、タイプミスをして9の代わりに"09"と書くことがあります。GCCは次のようなことを報告するかもしれませんが：`error: 無効な数字"9" は8進定数です`

また、8進数のシステムは、Javaではやや人気があります。IDAが印刷不可能な文字を含むJava文字列を表示すると、16進数ではなく8進数でエンコードされます。JAD Javaデコンパイラも同じように動作します。

除算能力

120のような10進数を見ると、最後の桁がゼロであるため、itfsを10で割り切れるとすぐに推論することができます。同様に、最後の2桁が0であるため、123400は100で割り切れる。同様に、16進数の0x1230は0x10（または16）で割り切れ、0x123000は0x1000（または4096）で割り切れる

バイナリ番号0b1000101000は0b1000（8）などで割り切れます。このプロパティは、メモリ内の一部のブロックのサイズがある境界に埋め込まれているかどうかを素早く認識するためによく使用されます。たとえば、PE12ファイルのセクションは、ほとんどの場合、0x41000、0x10001000など3つの16進ゼロで終わるアドレスで開始されます。これは、ほとんどすべてのPE¹²セクションが0x1000（4096）バイトの境界にパディングされているためです。

多精度算術演算と基数

多精度算術演算では膨大な数を使用でき、それぞれが数バイトで格納されます。たとえば、公開鍵と秘密鍵の両方のRSA鍵は、最大4096ビットに及んでいます。

¹²Portable Executable

[Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] において、我々は多バイト数で多倍精度の数値を格納するとき、整数を表すことができる $28 = 256$ の基数を有するものとして、各桁は対応するバイトに進む。同様に、複数の精度の数値を複数の32ビットの整数値に格納すると、各桁は32ビットの各スロットに移動し、この数値は基数232に格納されていると考えることができます。

非小数点の発音方法

非小数点の基数の数字は、通常、桁で数字によって発音されます。イチ・ゼロ・ゼロ・イチ・イチ。10や1000のような言葉は、小数点の基本システムとの混同を避けるために、通常は発音されません。

浮動小数点数

浮動小数点数を整数から区別するために、浮動小数点数は通常 0.0, 123.0 などの末尾に.0で書かれています。

第1.3節空関数

可能な限り単純な関数は、何もしない関数です。

Listing 1.1: **Japanese text placeholder**

```
void f()
{
    return;
};
```

コンパイルしましょう！

第1.3.1節x86

x86プラットフォーム上でGCCコンパイラとMSVCコンパイラの両方の生成物は次のとおりです。

Listing 1.2: 最適化 GCC/MSVC (アセンブリ出力)

```
f:
    ret
```

RET 命令のみです。callerに戻る命令です。

第1.3.2節ARM

Listing 1.3: 最適化 Keil 6/2013 (ARMモード) アセンブリ出力

```
f      PROC
      BX      lr
      ENDP
```

リターンアドレスはARM ISAのローカルスタックには保存されず、リンクレジスタに保存されるため、BX LR 命令は実行をそのアドレスにジャンプさせるため、[caller](#)への実行が効果的に戻ります。

第1.3.3節MIPS

レジスタの命名には、数値 (0~31) または擬似名 (V0~A0など) の2つの命名規則がMIPSの世界で使用されています。

以下のGCCアセンブリ出力は、レジスタを番号順にリストしています。

Listing 1.4: 最適化 GCC 4.4.5 (アセンブリ出力)

```
j      $31
nop
```

...[IDA¹³](#) は擬似名を使います

Listing 1.5: 最適化 GCC 4.4.5 (IDA)

```
j      $ra
nop
```

最初の命令は、実行フローを[caller](#)に返すジャンプ命令 (JまたはJR) で、\$31 (または \$RA) レジスタのアドレスにジャンプします。

これはARMの[LR¹⁴](#)に類似したレジスタです。

2番目の命令は[NOP¹⁵](#)で、何もしません。私たちは今それを無視することができます。

MIPS命令とレジスタ名についての注意

MIPSの世界では、レジスタと命令の名前は、伝統的に小文字で書かれています。しかし、一貫性を保つために、この本は大文字の使用で通します。

第1.3.4節実際の空関数

空の関数は役に立たないように見えますが、低レベルのコードでは頻繁に使用されます。

まず第一に、次のようなデバッグ機能でとてもポピュラーです。

Listing 1.6: C/C++ Code

```
void dbg_print (const char *fmt, ...)
{
#ifdef _DEBUG
    // open log file
    // write to log file
    // close log file
#endif
}
```

¹³ [Hex-Rays](#) によって開発されたインタラクティブなディスアセンブラ・デバッガ

¹⁴ Link Register

¹⁵ No Operation

```
};

void some_function()
{
    ...

    dbg_print ("we did something\n");

    ...
};
```

非デバッグビルドでは(greleasehのように)、DEBUG は定義されていないので、dbg_print() 関数は実行中に呼び出されているにもかかわらず、空になります。

同様に、ソフトウェア保護の一般的な方法は、合法的な顧客向けに1つのビルドを作成し、他にはデモビルドを作成することです。デモビルドには、この例のようにいくつかの重要な機能が欠けています。

Listing 1.7: C/C++ Code

```
void save_file ()
{
#ifdef DEMO
    // a real saving code
#endif
};
```

save_file() 関数は、ユーザーがメニューの File->Save をクリックすると呼び出すことができます。デモ版は、このメニュー項目を無効にしてお届けできますが、ソフトウェアクラッカーが有効にしても、役に立つコードのない空の関数だけが呼び出されます。

IDAは、nullsub_00、nullsub_01 などの名前でこのような機能をマークします。

第1.4節戻り値

もう1つの単純な関数は、単に定数値を返す関数です。

Listing 1.8: **Japanese text placeholder**

```
int f()
{
    return 123;
};
```

コンパイルしてみましょう。

第1.4.1節x86

ここでは、GCCコンパイラとMSVCコンパイラの両方でx86プラットフォーム上で（最適化を使用して）生成されるものを示します。

Listing 1.9: 最適化 GCC/MSVC (アセンブリ出力)

```
f:
    mov     eax, 123
    ret
```

命令はたった2つです：最初に値123を EAX レジスタに入れます。これは戻り値を格納するために慣例によって使用され、2つ目は`caller`に実行を返す RET です。

呼び出し元は EAX レジスタから結果を取得します。

第1.4.2節ARM

ARMプラットフォームにはいくつかの違いがあります。

Listing 1.10: 最適化 Keil 6/2013 (ARMモード) ASM Output

```
f      PROC
      MOV     r0,#0x7b ; 123
      BX      lr
      ENDP
```

ARMは関数の結果を返すためにレジスタ R0 を使用するため、123が R0 にコピーされます。

MOV は、x86とARMのISAの両方で命令の誤解を招く名前であることに注意する価値があります。

データは実際には移動されませんが、コピーされます。

第1.4.3節MIPS

以下のGCCアセンブリ出力は、レジスタを番号順にリストしています。

Listing 1.11: 最適化 GCC 4.4.5 (アセンブリ出力)

```
j      $31
li      $2,123          # 0x7b
```

...`IDA` は擬似名でリストします

Listing 1.12: 最適化 GCC 4.4.5 (IDA)

```
jr      $ra
li      $v0, 0x7B
```

\$2（または \$V0）レジスタは、関数の戻り値を格納するために使用されます。LI は「即時ロード」を表し、MOV に相当するMIPSです。

もう1つの命令は、実行フローを`caller`に返すジャンプ命令（JまたはJR）です。

ロード命令（LI）とジャンプ命令（JまたはJR）の位置が入れ替えられたのはなぜだろうか。これは、“branch delay slot” と呼ばれるRISC機能が原因です。

これが起こる理由は、いくつかのRISC ISAのアーキテクチャーでは奇抜であり、私たちの目的にとって重要ではありません。MIPSでは、ジャンプ命令または分岐命令に続く命令は、ジャンプ/分岐命令自体の前に実行される。

結果として、分岐命令は、直前に実行された命令で常に場所を入れ替える。

実際には、単に1（真）または0（偽）を返す関数はよく使われます。

標準のUNIXユーティリティである/bin/true と /bin/false の中でも最小のものがそれぞれ0と1を終了コードとして返します。（ゼロは終了コードとして通常成功を意味し、ゼロ以外はエラーを意味します）。

第1.5節ハローワールド!

[Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)]
という本の有名な例を使ってみましょう

Listing 1.13: C/C++ Code

```
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

第1.5.1節x86

MSVC

MSVC 2010でコンパイルしてみましょう。

```
cl 1.cpp /Fa1.asm
```

(/Fa オプションは、アセンブリリストファイルを生成するようにコンパイラに指示します)

Listing 1.14: MSVC 2010

```
CONST    SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST    ENDS
PUBLIC   _main
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG3830
    call    _printf
```

```

        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS

```

MSVCは、Intel構文でアセンブリリストを生成します。Intel構文とAT&T構文の違いについては、[1.5.1 on page 14](#)で説明します。

コンパイラは、1.exe にリンクされる 1.obj というファイルを生成了ました。私たちの場合、ファイルには CONST（データ定数用）と _TEXT（コード用）の2つのセグメントが含まれています。

C/C++の文字列 hello, world には、const char[][Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2] 型がありますが、独自の名前はありません。コンパイラは何らかの形で文字列を処理する必要があるため、内部名 \$SG3830 を定義します。

そのため、この例は次のように書き換えられます。

```

#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}

```

アセンブリリストに戻りましょう。わかるように、文字列は C/C++ 文字列の標準である NULL バイトで終了します。C/C++ 文字列の詳細は：[?? on page ??](#)

_TEXT というコードセグメントでは、main() 関数が1つしかありません。関数 main() は、プロローグコードで始まり、エピローグコードで終わります（ほぼすべての関数のように）¹⁶

関数のプロローグの後に、printf() 関数の呼び出しがあります。CALL _printf. 呼び出しの前に、PUSH 命令の助けを借りて、挨拶を含む文字列アドレス（またはそのポインタ）がスタックに置かれます。

printf() 関数が main() 関数に制御を返すと、文字列アドレス（またはそのポインタ）はまだスタック上にあります。もはや必要がないので、スタックポインタ（ESPレジスタ）を修正する必要があります。

ADD ESP, 4 は ESP レジスタ値に4を加算することを意味します。

なぜ4？これは32ビットプログラムなので、スタックを通過するアドレスには正確に4バイトが必要です。x64コードの場合は8バイト必要です。ADD ESP, 4 は POP register と事実上同等ですが、レジスタを使用しません¹⁷

¹⁶ プロローグとエピローグ関数についてのセクションでは、詳細を見ていきます ([1.6 on page 38](#))

¹⁷ CPU flags, however, are modified

同じ目的のために、インテルC++コンパイラのようなコンパイラの中には、ADD の代わりに POP ECX を発行するものもある（例えば、インテルC++コンパイラでコンパイルされているので、このようなパターンは Oracle RDBMS コードで見ることができる）。この命令はほとんど同じ効果を持ちますが、ECX レジスタの内容は上書きされます。インテルC++コンパイラは、この命令命令コードが ADD ESP, x（POP の場合は1バイト、ADD の場合は3バイト）よりも短いため、POP ECX を使用すると思われます。

Oracle RDBMS から ADD の代わりに POP を使用する例を次に示します。

Listing 1.15: Oracle RDBMS 10.2 Linux (app.o file)

.text:0800029A	push	ebx
.text:0800029B	call	qksfroChild
.text:080002A0	pop	ecx

printf() を呼び出した後、元の C/C++ コードには、main() 関数の結果として return 0 というステートメントが含まれています。

生成されたコードでは、これは命令 XOR EAX, EAX によって実装されます。

XOR は実際には「eXclusive OR」¹⁸ですが、コンパイラでは MOV EAX, 0 の代わりに使用されることがよくあります。もう少し短いオペコード（MOV の場合は5に対して XOR の場合は2バイト）であるからです。

一部のコンパイラは SUB EAX, EAX を出力します。これは、EAX の値を EAX の値から 差し引くことを意味します。それはどんな場合でもゼロになります。

最後の命令RETは、[caller](#)に制御を返します。通常、これは C/C++ [CRT](#)¹⁹コードであり、これは[OS](#)に制御を戻します。

GCC

LinuxのGCC 4.4.1コンパイラと同じ C/C++ コードをコンパイルしてみましょう：gcc 1.c -o 1 次に、[IDA](#) 逆アセンブラの助けを借りて、どのように main() 関数が作成されるのかを見ていきましょう。[IDA](#) は、MSVCと同様に、Intel-syntaxを使用します。²⁰

Listing 1.16: code in [IDA](#)

main	proc near
var_10	= dword ptr -10h
	push ebp
	mov ebp, esp
	and esp, 0FFFFFF0h
	sub esp, 10h
	mov eax, offset aHelloWorld ; "hello, world\n"
	mov [esp+10h+var_10], eax
	call _printf
	mov eax, 0

¹⁸[Wikipedia](#)

¹⁹C Runtime library

²⁰-S -masm=intel オプションを適用することで、Intel構文でGCCのアセンブリリストを生成させることもできます

	leave
	retn
main	endp

結果はほぼ同じです。helloのワールド文字列（データセグメントに格納されている）のアドレスは、最初にEAXレジスタにロードされ、スタックに保存されます。

さらに、関数プロローグには AND ESP, 0FFFFFFF0h があります。この命令は、ESP レジスタ値を16バイトの境界に揃えます。この結果、スタック内のすべての値が同じ方法で整列されます（処理中の値が4バイト境界または16バイト境界に整列したアドレスにある場合、CPUのパフォーマンスは向上します）。

SUB ESP, 10h はスタックに16バイトを割り当てます。しかし、私たちはこれから見るように、ここでは4つだけが必要です。

これは、割り当てられたスタックのサイズも16バイトの境界に揃えられているためです。

文字列アドレス（または文字列へのポインタ）は、PUSH 命令を使用せずにスタックに直接格納されます。var_10 はローカル変数であり、printf() の引数でもあります。それについて下記を読んでください。

printf() 関数が呼び出されます。

MSVCと異なり、GCCは最適化をオンにしないでコンパイルすると、より短いオペコードの代わりに MOV EAX, 0 を発行します。

最後の命令、LEAVE は、MOV ESP, EBP、および POP EBP 命令ペアと同等です。言い換えれば、この命令は**スタックポインタ** (ESP) を戻し、EBP レジスタを初期状態に戻す。これは、これらのレジスタ値 (ESP と EBP) を関数の始めに (MOV EBP, ESP / AND ESP, ... を実行することによって) 変更したので必要です。

GCC: AT&T構文

これをアセンブリ言語のAT&T構文でどのように表現できるかを見てみましょう。この構文は UNIXの世界ですっと人気があります。

Listing 1.17: let's compile in GCC 4.7.3

```
gcc -S 1_1.c
```

下記を得ます。

Listing 1.18: GCC 4.7.3

```
.file "1_1.c"
.section .rodata
.LC0:
.string "hello, world\n"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushl %ebp
```

```

.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl    %esp, %ebp
.cfi_def_cfa_register 5
andl    $-16, %esp
subl    $16, %esp
movl    $.LC0, (%esp)
call    printf
movl    $0, %eax
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
.LFE0:
.size    main, .-main
.ident   "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
.section .note.GNU-stack,"",@progbits

```

リストには、多くのマクロ（ドットで始まる部分）が含まれています。現時点では興味深いものではありません。

今のところ、簡単にするため、無視してもかまいません（C言語の文字列のように終端文字列をエンコードする `.string` マクロを除く）。それからこれを見てみましょう。²¹

Listing 1.19: GCC 4.7.3

```

.LC0:
.string "hello, world\n"
main:
    pushl    %ebp
    movl     %esp, %ebp
    andl     $-16, %esp
    subl     $16, %esp
    movl     $.LC0, (%esp)
    call     printf
    movl     $0, %eax
    leave
    ret

```

インテルとAT&T構文の主な違いのいくつかは次のとおりです。

- ソースオペランドとデスティネーションオペランドは逆の順序で記述されます。

インテル構文では：<命令> <デスティネーション・オペランド> <ソース・オペランド>

AT&T構文の場合：<命令> <ソースオペランド> <デスティネーションオペランド>

違いを覚えるのは簡単な方法です：インテル構文を扱うとき、オペランド間に等号(=)があり、AT&T-syntaxを扱うときに右矢印(→)があると想像してください。²²

²¹このGCCオプションは「不要な」マクロを削除するために使用できます： `-fno-asynchronous-unwind-tables`

²²ところで、いくつかのC標準関数（例えば、`memcpy()`、`strcpy()`）では、インテル構文と同じ方法で引数がリストされています。まず、デスティネーションメモリブロックへのポインタ、次にソースメモリブロックへのポインタです

- AT&T: レジスタ名の前にはパーセント記号 (%) を、数字の前にはドル記号 (\$) を書く必要があります。角カッコのかわりに丸カッコが使用されています。
- AT&T: オペランドサイズを定義する命令に接尾辞が追加されます
 - q — quad (64 bits)
 - l — long (32 bits)
 - w — word (16 bits)
 - b — byte (8 bits)

コンパイル結果に戻るには、IDA が表示した結果とほぼ同じです。微かな違いが1つあります。0FFFFFFF0h は \$-16 として表示されます。これは同じことです: 10進数の 16 は16進数で 0x10 です。-0x10 は 0xFFFFFFF0 に等しくなります (32ビットデータ型の場合)。

もう1つ: 戻り値は、XOR ではなく通常の MOV を使用して0に設定されます。MOV はレジスタに値をロードするだけです。誤ってつけられた名前です (データは移動せず、コピーされるため)。他のアーキテクチャでは、この命令の名前は「LOAD」または「STORE」などと同様です。

文字列のパッチ (Win32)

Hiewを使用して、実行可能ファイル内の “hello, world” 文字列を簡単に見つけることができます:

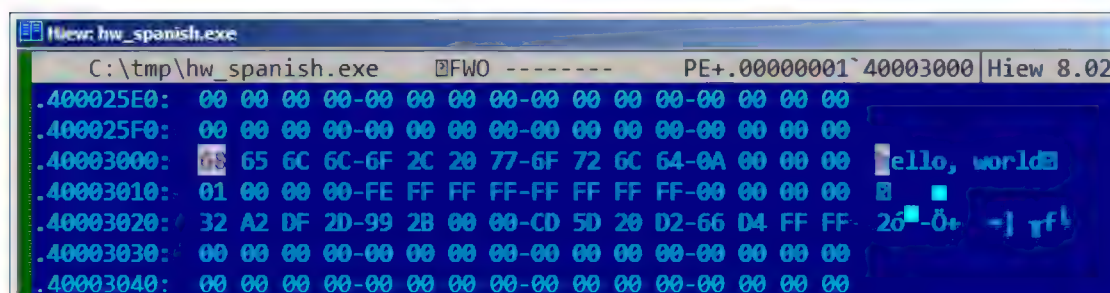


図 1.1: Hiew

メッセージをスペイン語に翻訳しようとすることができます

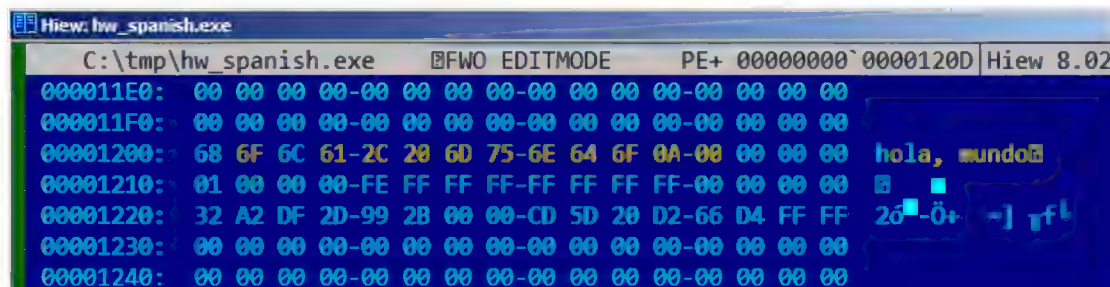


図 1.2: Hiew

スペイン語のテキストは英語より1バイト短くなっているのので、最後に0x0Aバイト (\n) に続けてNULLバイトを追加しました。

うまくいきました。

より長いメッセージを挿入する場合はどうすればよいですか？元の英語テキストの後には、ゼロバイトがいくつかあります。上書きできるかどうかはなんとも言えません：CRTコードのどこかで使われるかもしれないし、そうでないかもしれません。とにかく、自分が行っていることを本当に知っていれば、それらを上書きするだけです。

文字列のパッチ (Linux x64)

rada.re を使ってLinux x64実行ファイルにパッチを当ててみましょう

Listing 1.20: rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c

dennis@bigbox ~/tmp % radare2 a.out
-- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x004005c4 6865 6c6c 6f2c 2077 6f72 6c64 0000 0000 hello, world....
0x004005d4 011b 033b 3000 0000 0500 0000 1cfe ffff ...;0.....
0x004005e4 7c00 0000 5cfe ffff 4c00 0000 52ff ffff |...\...L...R...
0x004005f4 a400 0000 6cff ffff c400 0000 dcff ffff ....l.....
0x00400604 0c01 0000 1400 0000 0000 0000 017a 5200 .....zR.
0x00400614 0178 1001 1b0c 0708 9001 0710 1400 0000 .x.....
0x00400624 1c00 0000 08fe ffff 2a00 0000 0000 0000 .....*.....
0x00400634 0000 0000 1400 0000 0000 0000 017a 5200 .....zR.
0x00400644 0178 1001 1b0c 0708 9001 0000 2400 0000 .x.....$....
```

```

0x00400654  1c00 0000 98fd ffff 3000 0000 000e 1046 .....0.....F
0x00400664  0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422 ..J..w...?.;*3$
0x00400674  0000 0000 1c00 0000 4400 0000 a6fe ffff .....D.....
0x00400684  1500 0000 0041 0e10 8602 430d 0650 0c07 .....A....C..P..
0x00400694  0800 0000 4400 0000 6400 0000 a0fe ffff ....D...d.....
0x004006a4  6500 0000 0042 0e10 8f02 420e 188e 0345 e....B....B....E
0x004006b4  0e20 8d04 420e 288c 0548 0e30 8606 480e . . .B.(.H.0..H.

```

```

[0x004005c4]> oo+
File a.out reopened in read-write mode

```

```

[0x004005c4]> w hola, mundo\x00

```

```

[0x004005c4]> q

```

```

dennis@bigbox ~/tmp % ./a.out
hola, mundo

```

ここでは何が起きているの：私は/コマンドを使用して「hello」文字列を検索し、そのアドレスにカーソル（rada.re 用語でシーク）を設定します。次に、これが本当にその場所であることを確かめたい：px がダンプします。oo+ はrada.re を読み書きモードに切り替えます。w は現在のシーク時にASCII文字列を書き込みます。最後の\x00 に注意してください。これはNULLバイトです。q で終了します。

MS-DOS時代におけるソフトウェアのローカライズ

このやり方は、1980年代と1990年代にMS-DOSソフトウェアをロシア語に翻訳する一般的な方法でした。ロシア語の言葉や文章は、英語の文章と比べて通常若干長いので、ローカライズされたソフトウェアには奇妙な頭字語やとても読みにくい略語が含まれています。

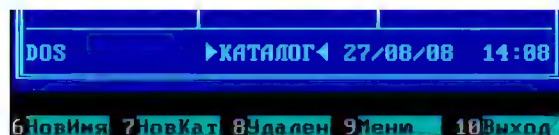


図 1.3: Japanese text placeholder

他の国の他の言語でも、この時代に起きていたことでしょう。

第1.5.2節x86-64

MSVC: x86-64

64ビットのMSVCも試してみましょう。

Listing 1.21: MSVC 2012 x64

```

$SG2989 DB      'hello, world', 0AH, 00H

main      PROC
sub       rsp, 40

```

```

    lea    rcx, OFFSET FLAT:$SG2989
    call   printf
    xor     eax, eax
    add     rsp, 40
    ret     0
main      ENDP

```

x86-64では、すべてのレジスタが64ビットに拡張されましたが、その名前には R-プレフィックスが付いています。スタックをあまり頻繁に使用しないように（言い換えると、外部メモリ/キャッシュにアクセスする頻度を減らす）、レジスタ (*fastcall*) ?? on page ?? を介して関数引数を渡す一般的な方法があります。すなわち、関数の引数の一部はレジスタに渡され、残りはスタックに渡されます。Win64では、4つの関数引数が RCX、RDX、R8、および R9 レジスタに渡されます。ここで見るものは `printf()` の文字列へのポインタはスタックにではなく、RCX レジスタに渡されます。ポインタは現在64ビットであるため、64ビットレジスタ（R-プレフィックスを持つ）に渡されます。ただし、下位互換性を保つために、E-接頭辞を使用して32ビットのパーツにアクセスすることは可能です。これは、RAX/EAX/AX/AL レジスタがx86-64のように見える方法です。

バイトの並び順							
第7	第6	第5	第4	第3	第2	第1	第0
RAX ^{x64}							
				EAX			
						AX	
						AH	AL

`main()` 関数は *int* 型の値を返します。これは C/C++ では32ビットのままで、下位互換性と移植性を向上させるため、関数終了時に RAX レジスタの代わりに EAX レジスタがクリアされる理由です。（すなわちレジスタの32ビットの部分）ローカルスタックには40バイトも割り当てられています。これは「シャドウスペース」と呼ばれます。これについては後で説明します：[1.10.2 on page 126](#)

GCC: x86-64

64ビット環境のLinuxでGCCを試してみましょう。

Listing 1.22: GCC 4.4.6 x64

```

.string "hello, world\n"
main:
    sub     rsp, 8
    mov     edi, OFFSET FLAT:.LC0 ; "hello, world\n"
    xor     eax, eax ; number of vector registers passed
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret

```

Linux、*BSDと Mac OS X は関数引数をレジスタに渡すためのメソッドも使います：
[Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application*

Binary Interface. AMD64 Architecture Processor Supplement, (2013)]²³

最初の6つの引数は、RDI, RSI, RDX, RCX, R8 および R9 レジスタに渡され、残りはスタックを介して渡されます。

そのため、文字列へのポインタは EDI (レジスタの32ビット部分) に渡されます。なぜ64ビット版の RDI を使用しないのでしょうか。

下位の32ビットレジスタ部分に何かを書き込む64ビットモードのすべての MOV 命令も上位32ビットをクリアすることが重要です (インテルマニュアル: 8.1.4 on page 564を参照)。つまり、MOV EAX, 011223344h は、上位ビットがクリアされるため、RAX に値を正しく書き込みます。

コンパイルされたオブジェクトファイル (.o) を開くと、すべての命令のオペコードも見ることができます。²⁴:

Listing 1.23: GCC 4.4.6 x64

.text:00000000004004D0	main	proc near
.text:00000000004004D0 48 83 EC 08	sub	rsp, 8
.text:00000000004004D4 BF E8 05 40 00	mov	edi, offset format ; "hello, world\n"
.text:00000000004004D9 31 C0	xor	eax, eax
.text:00000000004004DB E8 D8 FE FF FF	call	_printf
.text:00000000004004E0 31 C0	xor	eax, eax
.text:00000000004004E2 48 83 C4 08	add	rsp, 8
.text:00000000004004E6 C3	ret	
.text:00000000004004E6	main	endp

ご覧のように、0x4004D4 の EDI に書き込む命令は5バイトを占有します。64ビット値を RDI に書き込む同じ命令は7バイトを占有します。明らかに、GCCはいくらかのスペースを節約しようとしています。さらに、文字列を含むデータセグメントが4GiB以上のアドレスに割り当てられないことが保証されます。

また、printf() 関数呼び出しの前に EAX レジスタがクリアされていることがわかります。上記のABI²⁵標準によれば、使用されたベクトルレジスタの数はx86-64の*NIXシステムで EAX に渡されるためです。

アドレスのパッチ (Win64)

この例が/MD スイッチ (MSVCR*.DLL ファイルリンケージのために小さな実行可能ファイルを意味します) を使用してMSVC 2013でコンパイルされた場合、main() 関数が最初に来て簡単に見つかります

²³以下で利用可能 <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

²⁴これは オプション → ディスアセンブル → オペコードバイト数で有効になるはずですが

²⁵アプリケーション・バイナリー・インタフェース

```

Hiew: hw2.exe
C:\tmp\hw2.exe  FWO EDITMODE a64 PE+ 00000000`00000404 Hiew 8.02 (c)SEN
00000400: 4883EC28      sub    rsp, 028 ; '('
00000404: 48800DF51F0000 lea    rcx, [000002401]
00000408: FF15D7100000  call  q, [0000014E8]
00000411: 33C0          xor    eax, eax
00000413: 4883C428      add    rsp, 028 ; '('
00000417: C3           retn
00000418: 4883EC28      sub    rsp, 028 ; '('
0000041C: B8405A0000    mov    eax, 000005A4D ; ' ZH'
00000421: 663905D8EFFFF cmp    [-000000C00], ax
00000428: 7404          jz     00000042E
0000043F: 813850450000  cmp    d, [rax], 000004550 ; ' EP'
00000445: 75E3          jnz    00000042A
00000447: B908020000    mov    ecx, 000000208
0000044C: 66394818      cmp    [rax][018], cx
00000450: 75D8          jnz    00000042A
00000452: 33C9          xor    ecx, ecx
00000454: 83B884000000E cmp    d, [rax][000000084], 000000000
0000045B: 7609          jbe    000000466
0000045D: 3988F8000000  cmp    [rax][0000000F8], ecx
1 Help 2 3 4 Select 5 6 7 8 9 10 11

```

図 1.4: Hiew

実験として、アドレスを1ずつインクリメントすることができます：

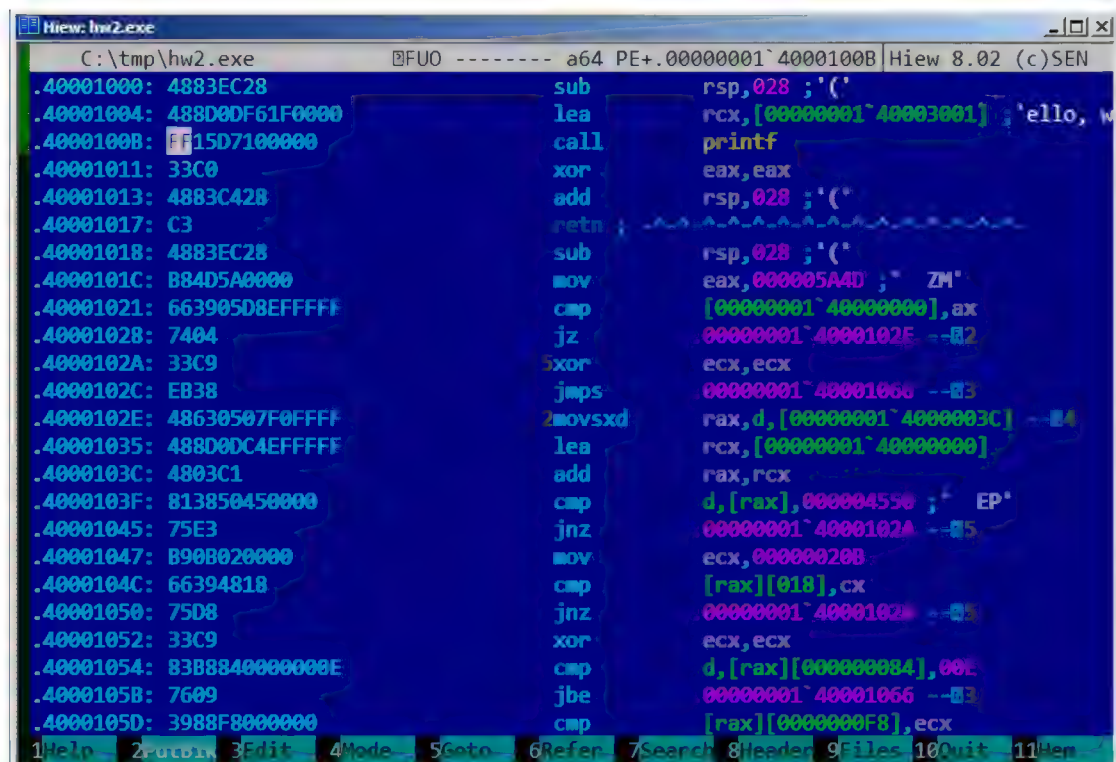


図 1.5: Hiew

Hiewは「ello, world」を示しています。そしてパッチが適用された実行可能ファイルを実行すると、この文字列が表示されます。

バイナリイメージから他の文字列を抜き取る (Linux x64)

Linux x64ボックスでGCC 5.4.0を使用して私たちの例をコンパイルしたときに得たバイナリファイルには、他の多くのテキスト文字列があります。ほとんどはインポートされた関数名とライブラリ名です。

objdumpを実行して、コンパイル済みファイルのすべてのセクションの内容を取得します。

```
$ objdump -s a.out
```

```
a.out:      file format elf64-x86-64
```

```
Contents of section .interp:
```

```
400238 2f6c6962 36342f6c 642d6c69 6e75782d /lib64/ld-linux-
400248 7838362d 36342e73 6f2e3200 x86-64.so.2.
```

```
Contents of section .note.ABI-tag:
```

```
400254 04000000 10000000 01000000 474e5500 .....GNU.
400264 00000000 02000000 06000000 20000000 ..... 
```



```
Contents of section .note.gnu.build-id:
 400274 04000000 14000000 03000000 474e5500 .....GNU.
 400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10 .F.x[.....^...
 400294 cf3f7ae4                .?z.

...
```

テキスト文字列「/lib64/ld-linux-x86-64.so.2」のアドレスを `printf()` に渡すのは問題ではありません

```
#include <stdio.h>

int main()
{
    printf(0x400238);
    return 0;
}
```

信じがたいですが、このコードは前述の文字列を表示します。

アドレスを `0x400260` に変更すると、「GNU」文字列が出力されます。このアドレスは、私の特定のGCCバージョン、GNUツールセットなどに当てはまります。あなたのシステムでは、実行ファイルは若干異なる場合があります、すべてのアドレスも異なります。また、このソースコードに/からコードを追加/削除すると、おそらくすべてのアドレスが前後に移動します。

第1.5.3節ARM

ARMプロセッサを使用した実験では、いくつかのコンパイラを使用しました。

- 組み込みの分野で人気があります：Keilリリース 2013/6
- LLVM-GCC 4.2コンパイラを搭載したApple Xcode 4.6.3 IDE ²⁶
- GCC 4.9 (Linaro) (ARM64用) は<http://www.linaro.org/projects/armv8/>で入手可能です。

特に記載がない場合、このマニュアルのすべてのケースで32ビットARMコード (ThumbおよびThumb-2モードを含む) が使用されます。64ビットARMについて話すときは、ARM64と呼びます。

非最適化 Keil 6/2013 (ARMモード)

Keilの例をコンパイルすることから始めましょう。

```
armcc.exe --arm --c90 -00 1.c
```

²⁶Apple Xcode 4.6.3は、オープンソースのGCCをフロントエンドコンパイラとLLVMコードジェネレータとして使用しています

armcc コンパイラはIntel構文でアセンブリリストを生成しますが、²⁷ それには高レベルのARMプロセッサに関連するマクロがありますが、「そのまま」の命令を見ることが重要ですので、*IDA* のコンパイル結果を見てみましょう。

Listing 1.24: 非最適化 Keil 6/2013 (ARMモード) *IDA*

```
.text:00000000          main
.text:00000000 10 40 2D E9      STMFD    SP!, {R4,LR}
.text:00000004 1E 0E 8F E2      ADR     R0, aHelloWorld ; "hello, world"
.text:00000008 15 19 00 EB      BL      __2printf
.text:0000000C 00 00 A0 E3      MOV     R0, #0
.text:00000010 10 80 BD E8      LDMFD    SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF:
main+4
```

この例では、各命令のサイズが4バイトであることを簡単に確認できます。実際、Thumb用ではなくARMモード用のコードをコンパイルしました。

最初の命令 `STMFD SP!, {R4,LR}`²⁸ は、2つのレジスタ (R4 と `LR`) の値をスタックに書き込むx86 `PUSH` 命令として機能します。

実際、*armcc* コンパイラの実出力リストには、簡略化のために実際に `PUSH {r4,lr}` 命令が示されています。しかしそれはかなり正確ではありません。`PUSH` 命令は、Thumbモードでのみ使用できます。したがって、物事をあまり混乱させないために、私たちは *IDA* でこれをやっています。

この命令は、最初に **SP!**³⁰ を **デクリメント**s して、新しいエントリがないスタック内の場所をポイントし、R4 および `LR` レジスタの値を変更された **SP!** に格納されたアドレスに保存します。

この命令 (Thumbモードの `PUSH` 命令のような) は、一度にいくつかのレジスタ値を保存することができ、非常に便利です。ところで、これはx86には同等の機能はありません。また、`STMFD` 命令は、**SP!**だけでなく、どのレジスタでも動作できるため、`PUSH` 命令の一般化 (機能拡張) であることにも注意してください。換言すれば、`STMFD` は、指定されたメモリアドレスにレジスタのセットを格納するために使用することもできます。

`ADR R0, aHelloWorld` 命令は、**PC!**³¹ レジスタの値を `hello, world` 文字列が配置されているオフセットに加算または減算します。ここでPCレジスタはどのように使用されるのですか？これは「位置独立コード」³²と呼ばれます。

このようなコードは、メモリ内の固定されていないアドレスで実行することができます。換言すれば、これは**PC!**相対アドレッシングである。

`ADR` 命令は、この命令のアドレスと文字列が配置されているアドレスとの間の差異を考慮する。この違い (オフセット) は、**OS**によってコードがロードされるアドレスに関係なく、常に同じになります。だから私たちが必要とするのは、現在の命令のアドレス (**PC!**から) を追加して、C文字列の絶対メモリアドレスを取得することだけです。

²⁷ 例えば ARMモードには `PUSH/POP` 命令がありません

²⁸ `STMFD`²⁹

³⁰ **SP!**

³¹ **PC!**

³² 関連セクションの詳細を読む: (?? on page ??)

BL __2printf³³命令は printf() 関数を呼び出します。この命令の仕組みは次のとおりです。

- BL 命令 (0xC) に続くアドレスをLRに格納する
- そのアドレスをPCレジスタに書き込むことによって、コントロールを printf() に渡します。

printf() の実行が終了すると、コントロールを返す必要がある場所に関する情報が必要です。各機能がLRレジスタに格納されたアドレスに制御を渡す理由です。

これはARMのような「純粋な」 RISCプロセッサとx86のようなCISC³⁴プロセッサとの違いです。リターンアドレスは通常スタックに格納されます。これについての詳細は、次のセクション (1.7 on page 39) を参照してください。

ところで、絶対32ビットのアドレスまたはオフセットは、24ビットのためのスペースしか有していないので、32ビット BL 命令では符号化することができない。思い出されるように、すべてのARMモード命令は4バイト (32ビット) のサイズを持ちます。したがって、それらは4バイトの境界アドレスにのみ配置することができます。これは、命令アドレスの最後の2ビット (常にゼロビット) が省略されることを意味する。要約すると、オフセットエンコーディングには26ビットがあります。これは $current_PC \pm \approx 32M$ をエンコードするのに十分です。

次に、MOV R0, #0³⁵命令は、R0 レジスタに0を書き込むだけです。これは、C関数が0を返し、戻り値が R0 レジスタに格納されるためです。

最後の命令 LDMFD SP!, R4, PC³⁶ スタック (または他のメモリ場所) から値をロードして R4 とPCに保存し、スタックポインタ SP!をincrementsします。ここで POP のように動作します。注意：最初の命令 STMFD は R4 とLRレジスタのペアをスタックに保存しましたが、R4 とPCは LDMFD の実行中にリストアされます。

すでにわかっているように、各関数が制御を返さなければならない場所のアドレスは、通常、LRレジスタに保存されます。最初の命令は、printf() を呼び出すときに main() 関数が同じレジスタを使用するため、その値をスタックに保存します。関数の終わりでは、この値を直接PCレジスタに書き込むことができ、したがって関数が呼び出された場所に制御を渡します。

main() は通常 C/C++ の主要な関数なので、コントロールはOSローダーやCRTのような点に返されます。

すべての機能を使用すると、関数の最後に BX LR 命令を省略できます。

DCB は、x86アセンブリ言語のDBディレクティブと同様に、バイトまたはASCII文字列の配列を定義するアセンブリ言語ディレクティブです。

非最適化 Keil 6/2013 (Thumbモード)

ThumbモードでKeilを使って同じ例をコンパイルしましょう。

```
armcc.exe --thumb --c90 -O0 1.c
```

³³Branch with Link

³⁴Complex Instruction Set Computing

³⁵MOVEの意味

³⁶LDMFD³⁷はSTMFDとは逆の命令です

IDAに入ってみましょう。

Listing 1.25: 非最適化 Keil 6/2013 (Thumbモード) + IDA

```
.text:00000000      main
.text:00000000 10 B5      PUSH    {R4,LR}
.text:00000002 C0 A0      ADR     R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9  BL     __2printf
.text:00000008 00 20      MOVVS  R0, #0
.text:0000000A 10 BD      POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF:
main+2
```

2バイト（16ビット）のオペコードを簡単に見つけることができます。これは既に述べたように、Thumbです。ただし、BL 命令は2つの16ビット命令で構成されています。これは、1つの16ビットオペコードの小さなスペースを使用している間に printf() 関数のオフセットをロードすることが不可能なためです。したがって、第1の16ビット命令はオフセットの上位10ビットをロードし、第2命令はオフセットの下位11ビットをロードする。

前述したように、Thumbモードの命令はすべて2バイト（または16ビット）のサイズです。これは、Thumb命令が奇妙なアドレスにあることはまったく不可能であることを意味します。上記を前提として、命令を符号化する間に最後のアドレスビットを省略することができます。

まとめると、BLThumb命令は $current_PC \pm \approx 2M$ のアドレスを符号化することができます。

関数内の他の命令については、PUSH と POP はここで説明した STMFD/LDMFD のように動作しますが、ここでは**SP!**レジスタのみが明示的に言及されていません。ADR は前の例と同様に動作します。MOVVS は、0を返すために R0 レジスタに0を書き込みます。

最適化 Xcode 4.6.3 (LLVM) (ARMモード)

最適化を有効にしない場合のXcode 4.6.3では、冗長なコードが多数生成されるため、命令カウントができるだけ小さい最適化された出力を検討し、コンパイラスイッチ -O3 を設定します。

Listing 1.26: 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

```
__text:000028C4      _hello_world
__text:000028C4 80 40 2D E9  STMFD    SP!, {R7,LR}
__text:000028C8 86 06 01 E3  MOV     R0, #0x1686
__text:000028CC 0D 70 A0 E1  MOV     R7, SP
__text:000028D0 00 00 40 E3  MOVT    R0, #0
__text:000028D4 00 00 8F E0  ADD     R0, PC, R0
__text:000028D8 C3 05 00 EB  BL      _puts
__text:000028DC 00 00 A0 E3  MOV     R0, #0
__text:000028E0 80 80 BD E8  LDMFD    SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0 DCB "Hello world!",0
```

命令 STMFD と LDMFD はもうよく知っていますね。

MOV 命令は、R0 レジスタに数値 0x1686 を書き込むだけです。これは「Hello world !」文字列を指すオフセットです。

R7 レジスタ ([iOS ABI Function Call Guide, (2010)]³⁸で標準化されている) はフレームポインタです。以下でもっとみてみましょう。

MOVT R0, #0 (MOVE Top) 命令は、レジスタの上位16ビットに0を書き込みます。ここでの問題点は、ARMモードの汎用MOV命令がレジスタの下位16ビットだけを書き込むことができることです。

ARMモードの命令オペコードはすべて32ビットに制限されています。もちろん、この制限はレジスタ間でのデータの移動には関係しません。そのため、上位ビット (16から31まで) に書き込むための命令 MOVT が追加されています。ただし、ここでの使用方法は冗長です。これは、MOV R0, #0x1686 命令がレジスタの上位部分をクリアしたためです。これはおそらくコンパイラの欠点です。

ADD R0, PC, R0 命令は、**PC!**の値を R0 の値に加算し、「Hello world!」文字列の絶対アドレスを計算します。すでにわかっているように、それは「位置独立コード」なので、この修正はここでは必須です。

BL 命令は printf() の代わりに puts() 関数を呼び出します。

LLVMは最初の printf() 呼び出しを puts() に置き換えました。確かに、唯一の引数を持つ printf() は、puts() とほぼ同じです。

ほとんどの場合、文字列に% で始まるprintf形式識別子が含まれていない場合にのみ、2つの関数が同じ結果を生成するためです。その場合、これらの2つの機能の効果は異なります³⁹

なぜコンパイラは printf() を puts() に置き換えたのでしょうか？おそらく puts() が高速であるためです。⁴⁰

これは、文字を% と一緒に比較することなく、文字をstdoutに渡すだけです。

次に、R0 レジスタを0に設定するための使い慣れた MOV R0, #0 命令があります。

最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

Xcode 4.6.3ではThumb-2のコードがデフォルトでは次のように生成されます。

Listing 1.27: 最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

__text:00002B6C		_hello_world
__text:00002B6C 80 B5	PUSH	{R7,LR}
__text:00002B6E 41 F2 D8 30	MOVW	R0, #0x13D8
__text:00002B72 6F 46	MOV	R7, SP
__text:00002B74 C0 F2 00 00	MOVT.W	R0, #0
__text:00002B78 78 44	ADD	R0, PC
__text:00002B7A 01 F0 38 EA	BLX	_puts
__text:00002B7E 00 20	MOVS	R0, #0
__text:00002B80 80 BD	POP	{R7,PC}

³⁸以下で利用可能 <http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf>

³⁹puts() は文字列の最後に改行記号 '\n' を必要としないので、ここでは見られません

⁴⁰ciselant.de/projects/gcc_printf/gcc_printf.html

...

```
__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld DCB "Hello world!",0xA,0
```

Thumbモードの BL と BLX 命令は、16ビット命令のペアとしてエンコードされています。Thumb-2では、これらの代理オペコードは、新しい命令がここで32ビット命令として符号化されるように拡張される。

これは、Thumb-2命令のオペコードが常に 0xFx または 0xE_x で始まることを考慮すると明らかです

しかし、IDA のリストでは、opcodeバイトはスワップされます。これは、ARMプロセッサの場合、命令は次のようにエンコードされるためです。最後のバイトが最初に来て、最初のバイトが来ると（ThumbおよびThumb-2モードの場合）、ARMモードの命令の場合、第1、第3、第2、そして最後に第1（異なるエンディアンのため）です。

つまり、バイトがIDAリストにどのように配置されているかです。

- ARMおよびARM64モードの場合：4-3-2-1;
- Thumbモードの場合：2-1;
- Thumb-2モードの16ビット命令の場合は2-1-4-3になります。

したがって、MOVW、MOVT.W および BLXX命令は 0xFx で始まります。

Thumb-2命令の1つは MOVW R0, #0x13D8 です。16ビット値を R0 レジスタの下部に格納し、上位ビットをクリアします。

また、MOVT.W R0, #0 は、前の例の MOVT と同様に動作し、Thumb-2でのみ動作します。BLX 命令は、BL の代わりにこの場合に使用されます。

違いは、RA⁴¹をLRレジスタに保存し、puts() 関数に制御を渡すことに加えて、プロセッサはThumb/Thumb-2モードからARMモード（またはその逆）にも切り替わります。

この命令は、制御が渡される命令が次のようになっているため、ここに配置されています（ARMモードでエンコードされています）。

```
__symbolstub1:00003FEC _puts          ; CODE XREF: _hello_world+E
__symbolstub1:00003FEC 44 F0 9F E5    LDR PC, =__imp_puts
```

これは本質的に、importsセクションに puts() のアドレスが書き込まれる場所へのジャンプです。

したがって、注意深い読者が質問するかもしれません：コードのどこに必要なところ
に puts () を呼び出すのはなぜですか？

非常にスペース効率が良いわけではないからです。

ほぼすべてのプログラムは外部のダイナミックライブラリ（WindowsではDLL、*NIXでは.so、Mac OS X では.dylib）を使用します。動的ライブラリには、標準のC関数puts ()を含む、頻繁に使用されるライブラリ関数が含まれています。

⁴¹リターンアドレス

実行可能バイナリファイル（Windows PE .exe、ELFまたはMach-O）には、インポートセクションが存在します。これは、外部モジュールからインポートされたシンボル（関数またはグローバル変数）のリストと、モジュール自体の名前です。

OSローダは、必要なすべてのモジュールをロードし、プライマリモジュールのインポートシンボルを列挙しながら、各シンボルの正しいアドレスを決定します。

私たちの場合、`_imp_puts` は、OSローダーが外部ライブラリに関数の正しいアドレスを格納するために使用する32ビットの変数です。次に、LDR 命令はこの変数から32ビットの値を読み込み、それを制御に渡して**PC!**レジスタに書き込みます。

したがって、この手順を完了するためにOSローダが必要とする時間を短縮するには、各シンボルのアドレスを専用の場所に1回だけ書き込むことをお勧めします。

さらに、すでにわかっているように、メモリアクセスなしで1つの命令だけを使用している間は、32ビットの値をレジスタにロードすることは不可能です。

したがって、最適な解決策は、ダイナミックライブラリに制御を渡し、次にThumbコードからこの短い1命令関数（いわゆる**thunk function**）にジャンプするという唯一の目的で、ARMモードで動作する別の関数を割り当てることです。

ところで、（ARMモード用にコンパイルされた）前の例では、コントロールはTTBLによって同じ**thunk function**に渡されます。ただし、プロセッサモードは切り替えられていません（したがって、命令ニーモニックに「X」がありません）。

thunk-functionsの追加情報

サンク関数は、誤った名前のために、明らかに理解するのが難しいです。1つのタイプのジャックのアダプターまたはコンバーターとして別のタイプのジャックに理解する最も簡単な方法です。たとえば、イギリスの電源プラグをアメリカのコンセントに差し込むことができるアダプタ、またはその逆。サンク関数はラッパーと呼ばれることもあります。

これらの関数についてもう少し詳しく説明します：

1961年にAlgol-60プロシージャコールの正式な定義に実際のパラメータをバインドする手段としてThunksを発明したP.Z. Ingermanによると、「アドレスを提供するコーディング」：仮パラメータの代わりに式を使用してプロシージャを呼び出すと、コンパイラは式を計算するサンクを生成し、結果のアドレスを何らかの標準の場所に残します。

…
マイクロソフトとIBMは、Intelベースのシステムでは、（プレティックセグメントレジスタと64Kアドレス制限付きの）「16ビット環境」とフラットアドレッシングとセミリアルメモリ管理を備えた「32ビット環境」を定義しています。この2つの環境は、同じコンピュータとOS上で動作することができます（Microsoftの世界では、Windows on Windowsの略です）。MSとIBMはどちらも、16ビットから32ビットへの変換プロセスを「サンク」と呼んでいます。Windows 95には、THUNK.EXEというツールがあります。これは「サンクコンパイラ」と呼ばれています。

([The Jargon File](#))

他の例としてLAPACK libraryがあります。FORTRANで書かれた“Linear Algebra PACKage”です。C/C++ 開発者もLAPACKを使いたいと思っていますが、C/C++ に書き直していくつかのバージョンを維持するのは難しいことです。ですから、C/C++ 環境から呼び出し可能な短いC関数があります。これは、順番にFORTRAN関数を呼び出し、他の何かを実行します。

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

また、そのような関数は“ラッパー”と呼ばれます。

ARM64

GCC

ARM64環境で、GCC 4.8.1を使用してサンプルをコンパイルしましょう。

Listing 1.28: 非最適化 GCC 4.8.1 + objdump

```
1 0000000000400590 <main>:
2 400590: a9bf7bfd stp x29, x30, [sp, #-16]!
3 400594: 910003fd mov x29, sp
4 400598: 90000000 adrp x0, 400000 <_init-0x3b8>
5 40059c: 91192000 add x0, x0, #0x648
6 4005a0: 97ffffa0 bl 400420 <puts@plt>
7 4005a4: 52800000 mov w0, #0x0 // #0
8 4005a8: a8c17bfd ldp x29, x30, [sp], #16
9 4005ac: d65f03c0 ret
10
11 ...
12
13 Contents of section .rodata:
14 400640 01000200 00000000 48656c6c 6f210a00 .....Hello!...
```

ARM64にはThumbモードとThumb-2モードはなく、ARMのみであるため、32ビット命令のみがあります。レジスタ数は2倍になります：?? on page ?? 64ビットレジスタは X-プレフィックスを持ち、32ビット部分は W-です。

STP 命令（ストアペア）は、スタック内の2つのレジスタ X29 と X30 を同時に保存します。

もちろん、この命令はメモリ内の任意の場所にこのペアを保存できますが、ここで**SP!**レジスタが指定されているため、ペアはスタックに保存されます。

ARM64レジスタは64ビットのレジスタで、それぞれ8バイトのサイズを持つため、2つのレジスタを保存するために16バイト必要です。

オペランドの後の感嘆符 ("!") は、最初に16が**SP!**から減算され、次にスタックに書き込まれるレジスタ・ペアの値であることを意味します。これは事前インデックスとも呼ばれます。事後インデックスと事前インデックスの違いについては、[1.30.2 on page 536](#) を読んでください。

したがって、より使い慣れたx86では、最初の命令は `PUSH X29` と `PUSH X30` のペアのアナログに過ぎません。X29 はARM64では**FP**⁴²として、**LR**では X30 として使用されているため、関数プロローグに保存され、関数エピローグで復元されます。

2番目の命令は X29（または**FP**）の**SP!**をコピーします。これは、関数スタックフレームを設定するために行われます。

`ADRP` 命令と `ADD` 命令は、最初の関数引数がこのレジスタに渡されるため、文字列「Hello!」のアドレスを X0 レジスタに入力するために使用されます。命令長は4バイトに制限されているため、レジスタに多数の命令を格納できる命令はありません。詳細は[1.30.3 on page 537](#)参照してください。したがって、いくつかの命令を利用する必要があります。最初の命令（`ADRP`）は、文字列が配置されている4KiBページのアドレスを X0 に書き込み、2番目の命令（`ADD`）は残りのアドレスをアドレスに追加するだけです。詳細については、[1.30.4 on page 540](#)を参照してください。

$0x400000 + 0x648 = 0x400648$ であり、このアドレスの `.rodata` データセグメントにある「Hello!」C文字列を参照してください。

`BL` 命令を使用して `puts()` を呼び出します。これについては既に説明しました：[1.5.3 on page 27](#)

`MOV` は W0 に0を書き込みます。W0 は64ビット X0 レジスタの下位32ビットです。

Japanese text placeholder	Japanese text placeholder
X0	
	W0

関数の結果は X0 を介して返され、`main()` は0を返します。これで、リターンされる結果がどのように準備されるのかがわかります。しかし、なぜ32ビットの部分を使用するのでしょうか？

ARM64の `int` データ型はx86-64の場合と同じように、互換性を高めるため、32ビットとなっています。

関数が32ビット `int` を返す場合は、X0 レジスタの下位32ビットのみを埋めなければなりません。

これを確認するために、この例を少し変更して再コンパイルしましょう。`main()` は64ビット値を返します：

Listing 1.29: `main()` returning a value of `uint64_t` type

```
#include <stdio.h>
#include <stdint.h>

uint64_t main()
{
    printf ("Hello!\n");
    return 0;
}
```

⁴²Frame Pointer

結果は同じですが、その行の MOV は次のようになります：

Listing 1.30: 非最適化 GCC 4.8.1 + objdump

4005a4:	d2800000	mov	x0, #0x0	// #0
---------	----------	-----	----------	-------

LDP (Load Pair) は X29 と X30 レジスタを復元します。

命令の後には感嘆符はありません。これは、値が最初にスタックからロードされ、次に **SP!** が16だけ増加したことを意味します。これは事後インデックスと呼ばれます。

ARM64 : RET という新しい命令が登場しました。これは BX LR と同様に機能し、特別なヒントビットのみが追加され、これが別のジャンプ命令ではなく関数からの戻りであることを **CPU** に通知するので、より最適に実行できます。

関数の単純さのために、GCCの最適化はまさに同じコードを生成します。

第1.5.4節MIPS

「グローバルポインタ」について少し

1つの重要なMIPSコンセプトは、「グローバルポインタ」です。既にわかっているように、各MIPS命令のサイズは32ビットなので、32ビットアドレスを1つの命令に組み込むことは不可能です（この例ではGCCのように対を使用しなければなりません読み込み）。ただし、1つの命令を使用してレジスタ $register - 32768 \dots register + 32767$ の範囲のアドレスからデータをロードすることは可能です（16ビットの符号付きオフセットを1つの命令でエンコードできるため）。したがって、この目的のためにいくつかのレジスタを割り当てて、最も多く使用されているデータの64KiB領域を割り当てることができます。この割り当てられたレジスタは「グローバルポインタ」と呼ばれ、64KiB領域の中央を指します。この領域には通常、`printf()` のようなインポートされた関数のグローバル変数とアドレスが含まれています。なぜなら、GCCの開発者は、関数のアドレスを得ることは2つではなく1つの命令の実行と同じくらい速くなければならないと判断したからです。ELFファイルでは、この64KiB領域は初期化されていないデータの場合は `.sbss`（「small **BSS**⁴³」）、初期化されたデータの場合は `.sdata`（「small data」）のセクションに部分的に配置されています。これはプログラマがどのデータを高速にアクセスしたいのかを選択して `.sdata` / `.sbss` に入れることを意味します。いくつかの古い学校のプログラマは、MS-DOSメモリモデル?? on page ??、またはすべてのメモリが64KiBブロックに分割されたXMS / EMSのようなMS-DOSメモリマネージャ。

この概念はMIPS特有のものではありません。少なくともPowerPCはこの手法も使用しています。

最適化 GCC

グローバルポインタの概念を示す次の例を考えてみましょう。

Listing 1.31: 最適化 GCC 4.4.5 (アセンブリ出力)

1 | \$LC0:

⁴³Block Started by Symbol

```

2 ; \000 is zero byte in octal base:
3     .ascii "Hello, world!\012\000"
4 main:
5 ; function prologue.
6 ; set the GP:
7     lui     $28,%hi(__gnu_local_gp)
8     addiu   $sp,$sp,-32
9     addiu   $28,$28,%lo(__gnu_local_gp)
10 ; save the RA to the local stack:
11     sw      $31,28($sp)
12 ; load the address of the puts() function from the GP to $25:
13     lw      $25,%call16(puts)($28)
14 ; load the address of the text string to $4 ($a0):
15     lui     $4,%hi($LC0)
16 ; jump to puts(), saving the return address in the link register:
17     jalr    $25
18     addiu   $4,$4,%lo($LC0) ; branch delay slot
19 ; restore the RA:
20     lw      $31,28($sp)
21 ; copy 0 from $zero to $v0:
22     move    $2,$0
23 ; return by jumping to the RA:
24     j       $31
25 ; function epilogue:
26     addiu   $sp,$sp,32 ; branch delay slot + free local stack

```

我々が見るように、\$GPレジスタは関数のプロローグでこの領域の中央を指すように設定されています。[RA](#)レジスタもローカルスタックに保存されます。printf() の代わりに puts() もここで使用されます。

puts() 関数のアドレスは、LW 命令（「Load Word」）を使用して \$25 にロードされます。LUI（「Load Upper Immediate」）と ADDIU（「Add Immediate Unsigned Word」）命令のペアを使用して、テキスト文字列のアドレスが \$4 にロードされます。LUI はレジスタの上位16ビット（したがって「命令名の上位ワード」）を設定し、ADDIU はアドレスの下位16ビットを加算します。

ADDIU は JALR に従います（まだ分岐遅延スロットを覚えていますか?）。レジスタ \$4 は \$A0 と呼ばれ、最初の関数引数を渡すために使用されます。⁴⁴

JALR（「Jump and Link Register」）は、[RA](#)の次の命令（LW）のアドレスを保存している間、\$25 レジスタ（puts() のアドレス）に格納されているアドレスにジャンプします。これはARMと非常によく似ています。ああ、重要なことの1つは、RAに保存されたアドレスは、次の命令のアドレスではないことです。（遅延スロットにあり、ジャンプ命令の前に実行されるため）したがって、PC+8 は JALR の実行中に[RA](#)に書き込まれます。私たちの場合、これは ADDIU の次の LW 命令のアドレスです。

20行目の LW（「Load Word」）は、ローカルスタックから[RA](#)を復元します（この命令は実際には関数のエピローグの一部です）。

22行目の MOVE は、\$0（\$ZERO）レジスタから \$2（\$V0）までの値をコピーします。

MIPSは定数レジスタを持ち、常に0を保持します。どうやら、MIPSの開発者たちは、実際

⁴⁴MIPSレジスタの表はappendixで見られます：?? on page ??

にはゼロがコンピュータプログラミングで最も忙しいという考えを思いついたので、ゼロが必要なたびに \$0レジスタを使用しましょう。

もう1つの興味深い事実は、MIPSにレジスタ間でデータを転送する命令がないことです。実際、MOVE DST, SRC は ADD DST, SRC, \$ZERO ($DST = SRC + 0$) です。これは同じです。明らかに、MIPS開発者はコンパクトなopcodeテーブルを用意したいと考えました。これは、各 MOVE 命令で実際の加算が行われることを意味しません。ほとんどの場合、CPUはこれらの疑似命令を最適化し、ALU⁴⁵は決して使用されません。

24行目の J は、RAのアドレスにジャンプします。これは、関数からの戻り値を効果的に実行しています。J の後の ADDIU は実際に J の前に実行されます（分岐遅延スロットを覚えていますか？）。そして関数のエピローグの一部です。ここに IDA によって生成されたリストもあります。この各レジスタには、独自の擬似名があります。

Listing 1.32: 最適化 GCC 4.4.5 (IDA)

```

1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10          = -0x10
4 .text:00000000 var_4          = -4
5 .text:00000000
6 ; function prologue.
7 ; set the GP:
8 .text:00000000                lui    $gp, (__gnu_local_gp >> 16)
9 .text:00000004                addiu   $sp, -0x20
10 .text:00000008                la     $gp, (__gnu_local_gp & 0xFFFF)
11 ; save the RA to the local stack:
12 .text:0000000C                sw     $ra, 0x20+var_4($sp)
13 ; save the GP to the local stack:
14 ; for some reason, this instruction is missing in the GCC assembly output:
15 .text:00000010                sw     $gp, 0x20+var_10($sp)
16 ; load the address of the puts() function from the GP to $t9:
17 .text:00000014                lw     $t9, (puts & 0xFFFF)($gp)
18 ; form the address of the text string in $a0:
19 .text:00000018                lui    $a0, ($LC0 >> 16) # "Hello, world!"
20 ; jump to puts(), saving the return address in the link register:
21 .text:0000001C                jalr   $t9
22 .text:00000020                la     $a0, ($LC0 & 0xFFFF) # "Hello,
    world!"
23 ; restore the RA:
24 .text:00000024                lw     $ra, 0x20+var_4($sp)
25 ; copy 0 from $zero to $v0:
26 .text:00000028                move   $v0, $zero
27 ; return by jumping to the RA:
28 .text:0000002C                jr     $ra
29 ; function epilogue:
30 .text:00000030                addiu   $sp, 0x20

```

15行目の命令は、GPの値をローカルスタックに保存します。この命令は、GCCの出力リストから不思議に見えます。GCCのエラーがあります⁴⁶ GPの値は実際に保存しなければなり

⁴⁵算術論理ユニット

⁴⁶明らかに、リストを生成する関数はGCCユーザーにとってあまり重要ではないので、修正されていないエラーがまだ存在するかもしれません

ません。データウィンドウ。puts() アドレスを含むレジスタは \$T9と呼ばれ、Tが前に付いたレジスタは「一時的」と呼ばれ、その内容は保持されない可能性があるためです。

非最適化 GCC

非最適化 GCCはもっと冗長です。

Listing 1.33: 非最適化 GCC 4.4.5 (アセンブリ出力)

```

1 $LC0:
2     .ascii  "Hello, world!\012\000"
3 main:
4     ; function prologue.
5     ; save the RA ($31) and FP in the stack:
6     addiu   $sp,$sp,-32
7     sw      $31,28($sp)
8     sw      $fp,24($sp)
9     ; set the FP (stack frame pointer):
10    move    $fp,$sp
11    ; set the GP:
12    lui     $28,%hi(__gnu_local_gp)
13    addiu   $28,$28,%lo(__gnu_local_gp)
14    ; load the address of the text string:
15    lui     $2,%hi($LC0)
16    addiu   $4,$2,%lo($LC0)
17    ; load the address of puts() using the GP:
18    lw      $2,%call16(puts)($28)
19    nop
20    ; call puts():
21    move    $25,$2
22    jalr    $25
23    nop    ; branch delay slot
24
25    ; restore the GP from the local stack:
26    lw      $28,16($fp)
27    ; set register $2 ($V0) to zero:
28    move    $2,$0
29    ; function epilogue.
30    ; restore the SP:
31    move    $sp,$fp
32    ; restore the RA:
33    lw      $31,28($sp)
34    ; restore the FP:
35    lw      $fp,24($sp)
36    addiu   $sp,$sp,32
37    ; jump to the RA:
38    j       $31
39    nop    ; branch delay slot

```

レジスタFPはスタックフレームへのポインタとして使用されることがわかります。3つのNOPも見てみましょう。2番目と3番目は分岐命令に従います。おそらく、GCCコンパイラは分岐命令の後に常に分岐遅延スロットのためにNOPを追加し、最適化がオンになって

いればそれらを削除するかもしれません。したがって、この場合、それらはここに残されます。

IDA のリストもあります：

Listing 1.34: 非最適化 GCC 4.4.5 (IDA)

```

1  .text:00000000 main:
2  .text:00000000
3  .text:00000000 var_10      = -0x10
4  .text:00000000 var_8      = -8
5  .text:00000000 var_4      = -4
6  .text:00000000
7  ; function prologue.
8  ; save the RA and FP in the stack:
9  .text:00000000          addiu   $sp, -0x20
10 .text:00000004          sw      $ra, 0x20+var_4($sp)
11 .text:00000008          sw      $fp, 0x20+var_8($sp)
12 ; set the FP (stack frame pointer):
13 .text:0000000C          move    $fp, $sp
14 ; set the GP:
15 .text:00000010          la      $gp, __gnu_local_gp
16 .text:00000018          sw      $gp, 0x20+var_10($sp)
17 ; load the address of the text string:
18 .text:0000001C          lui     $v0, (aHelloWorld >> 16) # "Hello,
    world!"
19 |.text:00000020          addiu   $a0, $v0, (aHelloWorld & 0xFFFF) #
    "Hello, world!"
20 ; load the address of puts() using the GP:
21 .text:00000024          lw      $v0, (puts & 0xFFFF)($gp)
22 .text:00000028          or      $at, $zero ; NOP
23 ; call puts():
24 .text:0000002C          move    $t9, $v0
25 .text:00000030          jalr    $t9
26 .text:00000034          or      $at, $zero ; NOP
27 ; restore the GP from local stack:
28 .text:00000038          lw      $gp, 0x20+var_10($fp)
29 ; set register $2 ($V0) to zero:
30 .text:0000003C          move    $v0, $zero
31 ; function epilogue.
32 ; restore the SP:
33 .text:00000040          move    $sp, $fp
34 ; restore the RA:
35 .text:00000044          lw      $ra, 0x20+var_4($sp)
36 ; restore the FP:
37 .text:00000048          lw      $fp, 0x20+var_8($sp)
38 .text:0000004C          addiu   $sp, 0x20
39 ; jump to the RA:
40 .text:00000050          jr      $ra
41 .text:00000054          or      $at, $zero ; NOP

```

興味深いことに、IDA は LUI/ADDIU 命令のペアを認識し、15行目の1つの LA (「Load Address」) 疑似命令に統合しました。この疑似命令のサイズは8バイトです。これは実際のMIPS命令ではなく、むしろ命令対のための便利な名前であるため、疑似命令 (またはマクロ) です。

もう1つのことは、**IDA** は**NOP**命令を認識していないことです。22行目、26行目、41行目です。OR \$AT, \$ZERO です。基本的に、この命令は、\$AT レジスタの内容にOR演算を0 (もちろんアイドル命令) で適用します。MIPSは他の多くの**ISA**と同様に、独立した**NOP**命令を持っていません。

スタックフレームの役割

テキスト文字列のアドレスはレジスタに渡されます。とにかくローカルスタックをセットアップする理由は？これは、printf() が呼び出されるため、レジスタ**RA**とGPの値をどこかに保存する必要があり、ローカルスタックがこの目的のために使用されているという事実にあります。これが **leaf function** であれば、関数のプロローグとエピローグを取り除くことができました。例：1.4.3 on page 10

最適化 **GCC: GDB**にロードしてみる

Listing 1.35: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw

root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:   lui      gp,0x42
0x00400644 <main+4>:   addiu   sp,sp,-32
0x00400648 <main+8>:   addiu   gp,gp,-30624
0x0040064c <main+12>:  sw      ra,28(sp)
0x00400650 <main+16>:  sw      gp,16(sp)
0x00400654 <main+20>:  lw      t9,-32716(gp)
0x00400658 <main+24>:  lui     a0,0x40
0x0040065c <main+28>:  jalr    t9
0x00400660 <main+32>:  addiu   a0,a0,2080
0x00400664 <main+36>:  lw      ra,28(sp)
0x00400668 <main+40>:  move    v0,zero
0x0040066c <main+44>:  jr      ra
0x00400670 <main+48>:  addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
```

```
(gdb) x/s $a0
0x400820:      "hello, world"
(gdb)
```

第1.5.5節結論

x86/ARMとx64/ARM64コードの主な違いは、文字列へのポインタが64ビット長になったことです。確かに、現代のCPUは64ビットになりました。これは、現代のアプリケーションではメモリの節約と大きな需要の両方があるからです。私たちは32ビットポインタよりもはるかに多くのメモリをコンピュータに追加することができます。そのため、すべてのポインタは64ビットになりました。

第1.5.6節練習問題

- <http://challenges.re/48>
- <http://challenges.re/49>

第1.6節関数のプロローグとエピローグ

関数プロローグは、関数の先頭にある一連の命令です。それはしばしば以下のコード断片のように見えます。

```
push    ebp
mov     ebp, esp
sub     esp, X
```

これらの命令が行うこと：EBP レジスタに値を保存し、EBP レジスタの値をESPの値に設定し、ローカル変数のためにスタック上に領域を割り当てます。

EBP の値は、関数実行の期間にわたって同じままであり、ローカル変数および引数アクセスに使用されます。同じ目的のために ESP を使うことができますが、時間の経過とともに変化するので、この方法はあまり便利ではありません。

関数のエピローグは、スタック内の割り当てられた領域を解放し、EBP レジスタの値を初期状態に戻し、制御フローをcallerに返します。

```
mov     esp, ebp
pop     ebp
ret     0
```

関数のプロローグとエピローグは、通常、逆アセンブラで関数の区切りとして検出されます。

第1.6.1節再帰

エピローグとプロローグは、再帰のパフォーマンスに悪影響を及ぼします。

この本の再帰の詳細は下記を参照: ?? on page ??

第1.7節 スタック

スタックは、コンピュータサイエンスにおける最も基本的なデータ構造の1つです。⁴⁷ AKA⁴⁸ LIFO⁴⁹.

技術的には、それは、プロセスメモリ内のメモリのブロックであり、x86またはx64の ESP または RSP レジスタ、またはARMの**SP!**レジスタをそのブロック内のポインタとして使用します。

最も頻繁に使用されるスタックアクセス命令は、PUSH と POP (x86およびARM Thumbモードの両方) です。PUSH は、32ビットモード (または64ビットモードでは8) で ESP/RSP/**SP!** 4を減算し、その単独オペランドの内容を ESP/RSP/**SP!**が指すメモリアドレスに書き込みます。

POP は逆の操作です：**SP!**が指し示すメモリ位置からデータを取り出し、命令オペランド (しばしばレジスタ) にロードし、**スタックポインタ**に4 (または8) を追加します。

スタック割り当ての後、**スタックポインタ**はスタックの一番下を指します。PUSH は**スタックポインタ**を減らし、POP はそれを増やします。スタックの最下部は実際にスタックブロックに割り当てられたメモリの先頭にあります。それは奇妙に見えますが、それはそうです。

ARMは降順スタックと昇順スタックの両方をサポートしています。

例えば、**STMTFD/LDMFD**、**STMED**⁵⁰/**LDMED**⁵¹命令は、降順のスタックを扱うことを意図しています (下位に向かって、高いアドレスから始まり、低いアドレスに進む)。**STMFA**⁵²/**LDMFA**⁵³、**STMEA**⁵⁴/**LDMEA**⁵⁵命令は、昇順のスタックを扱うことを意図しています (上位アドレスから始まり、上位アドレスに向かって進みます)。

第1.7.1節 スタックはなぜ後方に進むのか

直感的には、他のデータ構造と同様に、スタックが上方に、すなわちより高いアドレスに向かって成長すると考えるかもしれません。

スタックが後方に成長する理由はおそらく歴史的なものです。コンピュータが大きくて部屋全体を占有していた時代、メモリを2つの部分に分けるのは簡単でした。1つは **ヒープ** 用、もう1つはスタック用です。もちろん、プログラムの実行中に**ヒープ**とスタックがどれだけ大きくなるかは不明であったため、この解決策は最も簡単でした。

⁴⁷wikipedia.org/wiki/Call_stack

⁴⁸ 別名

⁴⁹ 後入れ先出し

⁵⁰ Store Multiple Empty Descending (ARM命令)

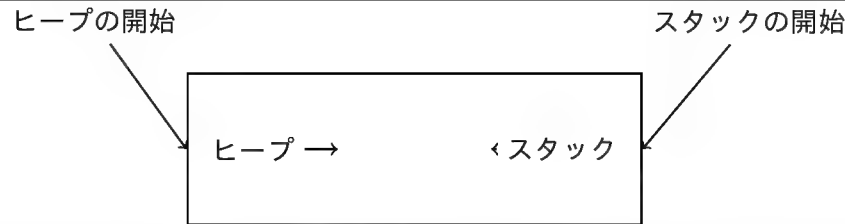
⁵¹ Load Multiple Empty Descending (ARM命令)

⁵² Store Multiple Full Ascending (ARM命令)

⁵³ Load Multiple Full Ascending (ARM命令)

⁵⁴ Store Multiple Empty Ascending (ARM命令)

⁵⁵ Load Multiple Empty Ascending (ARM命令)



[D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)]⁵⁶では、以下のように書かれています。

画像のユーザコア部分は、3つの論理セグメントに分割される。プログラムテキストセグメントは仮想アドレス空間の位置0で始まります。実行中、このセグメントは書き込み保護されており、同じプログラムを実行しているすべてのプロセス間でこのセグメントが共有されます。仮想アドレス空間のプログラムテキストセグメントの上の最初の8Kバイト境界では、共有されない書き込み可能なデータセグメントが開始されます。このデータセグメントのサイズはシステムコールによって拡張されます。仮想アドレス空間の最上位アドレスから始まるスタックセグメントは、ハードウェアのスタックポインタが変動すると自動的に下に向かって成長します。

これは、一部の学生が1つのノートブックを使用して2つの講義ノートを書く方法を思い出させます。最初の講義のノートはいつものように書かれ、2つ目のノートはノートブックの最後からそれを反転させて書き込まれます。空き領域がない場合に、ノートはその間のどこかで互いに会うことになります。

第1.7.2節スタックは何に使用されるか

関数のリターンアドレスを保存する

x86

CALL 命令で別の関数を呼び出すと、CALL 命令の直後のポイントのアドレスがスタックに保存され、CALL オペランドのアドレスへの無条件ジャンプが実行されます。

CALL 命令は、PUSHの PUSH address_after_call / JMP operand 命令対に相当する。

RET はスタックから値を取り出し、ジャンプします。これは POP tmp / JMP tmp 命令の対に相当します。

スタックのオーバーフローは簡単です。永遠の再帰を実行するだけです：

```
void f()
{
    f();
};
```

MSVC 2008が問題をレポートします：

⁵⁶以下で利用可能 [URL](#)


```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for x
  ↳ 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, x
  ↳ function will cause runtime stack overflow
```

...しかし、正しいコードを生成します。

```
?f@@YAXXZ PROC                                ; f
; Line 2
    push    ebp
    mov     ebp, esp
; Line 3
    call    ?f@@YAXXZ                        ; f
; Line 4
    pop     ebp
    ret     0
?f@@YAXXZ ENDP                                ; f
```

...また、コンパイラ最適化 (/Ox オプション) を有効にすると、最適化されたコードはスタックをオーバーフローせず、代わりに正しく⁵⁷動作します。

```
?f@@YAXXZ PROC                                ; f
; Line 2
$LL3@f:
; Line 3
    jmp     SHORT $LL3@f
?f@@YAXXZ ENDP                                ; f
```

GCC 4.4.1はどちらの場合も問題の警告を出さずに同様のコードを生成します。

ARM

また、ARMプログラムはスタックを使用してリターンアドレスを保存しますが、別の方法でスタックを使用します。「ハローワールド!」(1.5.3 on page 23) で述べたように、RAはLR (link register) に保存されます。ただし、別の関数を呼び出してもう一度LRレジスタを使用する必要がある場合は、その値を保存する必要があります。通常、関数プロローグに保存されます。

多くの場合、PUSH R4-R7, LR のような命令が、エピローグで POP R4-R7, PC とともに見られます。したがって、関数で使われるレジスタ値は、LRを含めてスタックに保存されます。

それにもかかわらず、ある関数が他の関数を呼び出すことがなければ、RISCの用語ではそれを *leaf function*⁵⁸ と呼びます。その結果、リーフ関数はLRレジスタを保存しません

⁵⁷ この皮肉

⁵⁸ infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

(LRレジスタを変更しないため)。このような関数が小さく、少数のレジスタを使用する場合は、スタックをまったく使用しないことがあります。したがって、スタックを使用せずにリーフ関数を呼び出すことができます。⁵⁹ これは、外部RAMがスタックに使用されないため、古いx86マシンよりも高速になる可能性があります。これは、スタックのメモリがまだ割り当てられていない状況または利用できません。

リーフ関数のいくつかの例：[1.10.3 on page 130](#), [1.10.3 on page 130](#), [1.275 on page 382](#), [1.291 on page 404](#), [1.21.5 on page 404](#), [1.185 on page 257](#), [1.183 on page 254](#), [1.202 on page 277](#).

関数の引数を渡す

x86でパラメータを渡す最も一般的な方法は、「cdecl」です。

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

callee関数はスタックポインタを介して引数を取得します。

したがって、`f()` 関数の最初の命令が実行される前に、引数の値がスタックにどのように格納されているかがわかります。

ESP	return address
ESP+4	引数#1, IDA にマークする <code>arg_0</code>
ESP+8	引数#2, IDA にマークする <code>arg_4</code>
ESP+0xC	引数#3, IDA にマークする <code>arg_8</code>
...	...

他の呼び出し規約の詳細については、セクション ([?? on page ??](#)) も参照してください。

ちなみに、**callee**関数には、渡された引数の数に関する情報はありません。(printf() のような) 可変数の引数を持つC関数は、フォーマット文字列指定子 (% 記号で始まる) を使ってその数を決定します。

私たちが次のように書くとして。

```
printf("%d %d %d", 1234);
```

`printf()` は1234を出力し、次にそのスタックの隣にある2つの乱数⁶⁰を出力します。

だから、`main()` 関数を宣言する方法はあまり重要ではありません：`main()` `main(int argc, char *argv[])` または `main(int argc, char *argv[], char *envp[])` のいずれかです。

実際、**CRT**コードは `main()` を以下のように呼び出しています：

⁵⁹いくつかの時間前、PDP-11とVAXでは、CALL命令（他の関数を呼び出す）は高価でした。実行時間の50%までが費やされる可能性があるため、小さな機能を多数持つことは[anti-pattern](#) [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II]

⁶⁰厳密な意味でランダムではなく、むしろ予測不可能: [1.7.4 on page 49](#)

```

push envp
push argv
push argc
call main
...

```

引数なしで `main()` を `main()` として宣言すると、`main()` はスタックにまだ残っていますが使用されません。`main()` を `main(int argc, char *argv[])` として宣言すると、最初の2つの引数を使用することができ、3つ目の引数は関数の「不可視」のままになります。さらに、`main(int argc)` を宣言することも可能です。これは動作します。

引数を渡す別の方法

プログラマがスタックを介して引数を渡すことは何も必要ではないことは注目に値する。それは要件ではありません。スタックをまったく使用せずに他の方法を実装することもできます。

アセンブリ言語初心者の中でやや普及している方法は、グローバル変数を介して引数を渡すことです

Listing 1.36: Assembly code

```

...

mov     X, 123
mov     Y, 456
call    do_something

...

X       dd      ?
Y       dd      ?

do_something proc near
; take X
; take Y
; do something
retn
do_something endp

```

しかし、このメソッドには明白な欠点があります。`do_something()` 関数は、独自の引数をzapする必要があるため、再帰的に（または別の関数を介して）呼び出すことはできません。ローカル変数を使った同じ話：グローバル変数でそれらを保持すると、関数は自分自身を呼び出すことができませんでした。また、これはスレッドセーフ⁶¹ではありません。このような情報をスタックに格納する方法は、これをより簡単にします。多くの関数の引数や値、スペースを確保できます。

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] は、IBM System/360上で特に便利な奇妙なスキームについても言及しています。

⁶¹正しく実装され、各スレッドは独自の引数/変数を持つ独自のスタックを持ちます

MS-DOSには、レジスタを介してすべての関数引数を渡す方法がありました。たとえば、古代16ビットMS-DOSの“Hello, world!” コードのコードです。

```
mov dx, msg      ; address of message
mov ah, 9        ; 9 means "print string" function
int 21h          ; DOS "syscall"

mov ah, 4ch      ; "terminate program" function
int 21h          ; DOS "syscall"

msg db 'Hello, World!\$'
```

これは、?? on page ??のメソッドと非常によく似ています。また、Linuxのsyscalls ((?? on page ??)) とWindowsを呼び出すのと非常によく似ています。

MS-DOS関数がブール値（すなわち単一ビット、通常はエラー状態を示す）を返す場合、CFフラグがしばしば使用されます。

例えば：

```
mov ah, 3ch      ; create file
lea dx, filename
mov cl, 1
int 21h
jc error
mov file_handle, ax
...
error:
...
```

エラーの場合、CFフラグが立てられます。それ以外の場合は、新しく作成されたファイルのハンドルが AX を介して返されます。

このメソッドは、アセンブリ言語プログラマによって引き続き使用されます。Windows Research Kernelのソースコード（Windows 2003と非常に似ています）では、次のようなものが見つかります

(ファイル *base/ntos/ke/i386/cpu.asm*)

```
public Get386Stepping
Get386Stepping proc

    call MultiplyTest          ; Perform multiplication test
    jnc short G3s00            ; if nc, muttest is ok
    mov ax, 0
    ret

G3s00:
    call Check386B0            ; Check for B0 stepping
    jnc short G3s05            ; if nc, it's B1/later
    mov ax, 100h               ; It is B0/earlier stepping
    ret

G3s05:
    call Check386D1            ; Check for D1 stepping
    jc short G3s10             ; if c, it is NOT D1
```

```

        mov     ax, 301h                ; It is D1/later stepping
        ret

G3s10:
        mov     ax, 101h                ; assume it is B1 stepping
        ret

        ...

MultiplyTest proc

        xor     cx,cx                    ; 64K times is a nice round number
mlt00:   push    cx
        call    Multiply                 ; does this chip's multiply work?
        pop     cx
        jc      short mltx               ; if c, No, exit
        loop    mlt00                   ; if nc, YEs, loop to try again
        clc

mltx:
        ret

MultiplyTest endp

```

ローカル変数記憶域

関数は、スタックの底に向かって**スタックポインタ**を減らすだけで、ローカル変数のためにスタックに領域を割り当てることができます。

したがって、どれだけ多くのローカル変数が定義されていても、非常に高速です。スタックにローカル変数を格納する必要もありません。あなたは好きな場所にローカル変数を格納することができますが、伝統的にはこれがどのように行われています。

x86: `alloca()` 関数

`alloca()` 関数に注目することは重要です ⁶² この関数は `malloc()` のように動作しますが、スタックに直接メモリを割り当てます。関数のエピローグ (1.6 on page 38) は ESP を初期状態に戻し、割り当てられたメモリは単に破棄されるため、割り当てられたメモリチャンクは `free()` 関数呼び出しで解放する必要はありません。`alloca()` がどのように実装されているかは注目に値する。簡単に言えば、この関数は必要なバイト数だけスタック底部に向かって ESP を下にシフトさせ、割り当てられたブロックへのポインタとして ESP を設定します。

やってみましょう。

```

#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif

```

⁶²MSVCでは、関数の実装は C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel の `alloca16.asm` と `chkstk.asm` にあります

```
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

`_snprintf()` 関数は `printf()` と同じように動作しますが、結果を [stdout](#) (ターミナルやコンソールなど) にダンプする代わりに、`buf` バッファに書き込みます。`puts()` 関数は `buf` の内容を [stdout](#) にコピーします。もちろん、これらの2つの関数呼び出しは1つの `printf()` 呼び出しで置き換えることができますが、小さなバッファの使用法を説明する必要があります。

MSVC

コンパイルしてみましょう (MSVC 2010で)

Listing 1.37: MSVC 2010

```
...

mov     eax, 600    ; 00000258H
call    __alloca_probe_16
mov     esi, esp

push    3
push    2
push    1
push    OFFSET $SG2672
push    600        ; 00000258H
push    esi
call    __snprintf

push    esi
call    _puts
add     esp, 28

...
```

`alloca()` の唯一の引数は EAX 経由で (スタックにプッシュするのではなく) 渡されます。
63

⁶³`alloca ()` はコンパイラ組み込み関数 ((?? on page ??)) ではなく、通常の関数です。[MSVC⁶⁴](#) の `alloca()` の実装には、割り当てられたメモリから読み込むコードが含まれているため、OSが物理メモリをVM領域にマップするために、コード内の命令が数個ではなく別々の関数を必要とする理由の1つです。`alloca()` 呼び出しの後、ESPは600バイトのブロックを指し、`buf` 配列のメモリとして使用できます。

GCC + インテル構文

GCC 4.4.1は、外部関数を呼び出すことなく同じことを行います

Listing 1.38: GCC 4.7.3

```
.LC0:
    .string "hi! %d, %d, %d\n"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 660
    lea     ebx, [esp+39]
    and     ebx, -16                ; align pointer by 16-byte border
    mov     DWORD PTR [esp], ebx    ; s
    mov     DWORD PTR [esp+20], 3
    mov     DWORD PTR [esp+16], 2
    mov     DWORD PTR [esp+12], 1
    mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov     DWORD PTR [esp+4], 600 ; maxlen
    call    _snprintf
    mov     DWORD PTR [esp], ebx    ; s
    call    puts
    mov     ebx, DWORD PTR [ebp-4]
    leave
    ret
```

GCC + AT&T構文

同じコードをAT&T構文で見てください

Listing 1.39: GCC 4.7.3

```
.LC0:
    .string "hi! %d, %d, %d\n"
f:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $660, %esp
    leal    39(%esp), %ebx
    andl    $-16, %ebx
    movl    %ebx, (%esp)
    movl    $3, 20(%esp)
    movl    $2, 16(%esp)
    movl    $1, 12(%esp)
    movl    $.LC0, 8(%esp)
    movl    $600, 4(%esp)
    call    _snprintf
    movl    %ebx, (%esp)
    call    puts
```

```

movl    -4(%ebp), %ebx
leave
ret

```

コードは前のリストと同じです。

ちなみに、`movl $3, 20(%esp)` は、Intel構文の `mov DWORD PTR [esp+20], 3` に対応しています。AT&Tの構文では、アドレス指定メモリのレジスタ+オフセット形式は `offset(%register)` のように見えます。

(Windows) SEH

SEH⁶⁵レコードはスタックにも格納されます（存在する場合）。それについてもっと読む：[\(5.2.1 on page 556\)](#)

バッファオーバーフロー保護

詳細はこちら [\(1.20.2 on page 332\)](#)

スタック内のデータの自動解放

おそらく、ローカル変数とSEHレコードをスタックに格納する理由は、スタックポインタを修正するための命令を1つだけ使用して（通常は `ADD` です）、関数が終了すると自動的に解放されるからです。関数の引数は、関数の終わりに自動的に割り当て解除されます。対照的に、ヒープに格納されているものはすべて明示的に割り当て解除する必要があります。

第1.7.3節典型的なスタックレイアウト

最初の命令を実行する前の、関数の開始時の32ビット環境での典型的なスタックレイアウトは次のようになります。

...	...
ESP-0xC	ローカル変数#2, IDA にマークする <code>var_8</code>
ESP-8	ローカル変数#1, IDA にマークする <code>var_4</code>
ESP-4	saved value of EBP
ESP	リターンアドレス
ESP+4	引数#1, IDA にマークする <code>arg_0</code>
ESP+8	引数#2, IDA にマークする <code>arg_4</code>
ESP+0xC	引数#3, IDA にマークする <code>arg_8</code>
...	...

⁶⁵Structured Exception Handling

第1.7.4節スタックのノイズ

ある人が何かがランダムに見えると言うとき、実際には、その中に何らかの規則性を見ることができないということです

Stephen Wolfram, A New Kind of Science.

多くの場合、この本では「ノイズ」や「ガベージ」の値がスタックやメモリに記述されています。彼らはどこから来たのか？これらは、他の関数の実行後にそこに残っているものです。短い例：

```
#include <stdio.h>

void f1()
{
    int a=1, b=2, c=3;
};

void f2()
{
    int a, b, c;
    printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
    f1();
    f2();
};
```

コンパイルすると ...

Listing 1.40: 非最適化 MSVC 2010

```
$SG2752 DB      '%d, %d, %d', 0aH, 00H

_c$ = -12      ; size = 4
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
_f1 PROC
    push      ebp
    mov       ebp, esp
    sub       esp, 12
    mov       DWORD PTR _a$[ebp], 1
    mov       DWORD PTR _b$[ebp], 2
    mov       DWORD PTR _c$[ebp], 3
    mov       esp, ebp
    pop       ebp
    ret       0
_f1 ENDP

_c$ = -12      ; size = 4
_b$ = -8       ; size = 4
```

```

_a$ = -4      ; size = 4
_f2 PROC
    push     ebp
    mov     ebp, esp
    sub     esp, 12
    mov     eax, DWORD PTR _c$[ebp]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp]
    push    edx
    push    OFFSET $SG2752 ; '%d, %d, %d'
    call    DWORD PTR __imp__printf
    add     esp, 16
    mov     esp, ebp
    pop     ebp
    ret     0
_f2 ENDP

_main PROC
    push    ebp
    mov     ebp, esp
    call    _f1
    call    _f2
    xor     eax, eax
    pop     ebp
    ret     0
_main ENDP

```

コンパイラは少し不満そうです...

```

c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for x
  ↳ 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c'  ↳
  ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b'  ↳
  ↳ used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a'  ↳
  ↳ used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:st.exe
st.obj

```

しかし、コンパイルされたプログラムを実行すると ...

```

c:\Polygon\c>st
1, 2, 3

```

ああ、なんて奇妙なんでしょう！我々は `f2()` に変数を設定しませんでした。これらは「ゴースト」値であり、まだスタックに入っています。

サンプルを OllyDbg にロードしましょう。

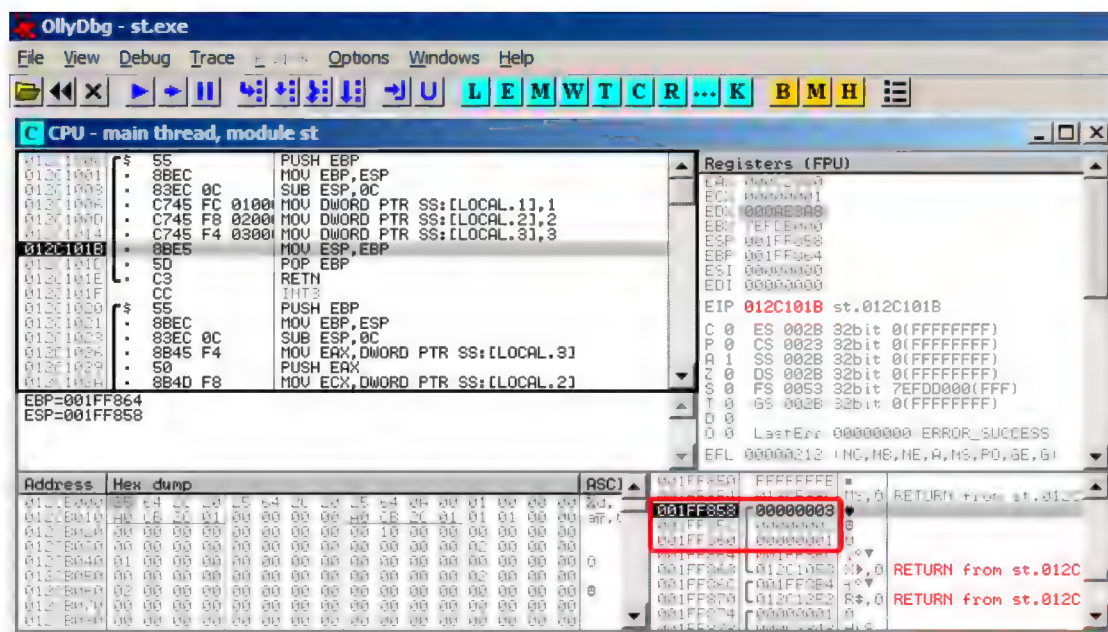


図 1.6: OllyDbg: `f1()`

`f1()` が変数 a 、 b 、 c を代入すると、その値はアドレス `0x1FF860` に格納されます。

そして f2() が実行されるとき：

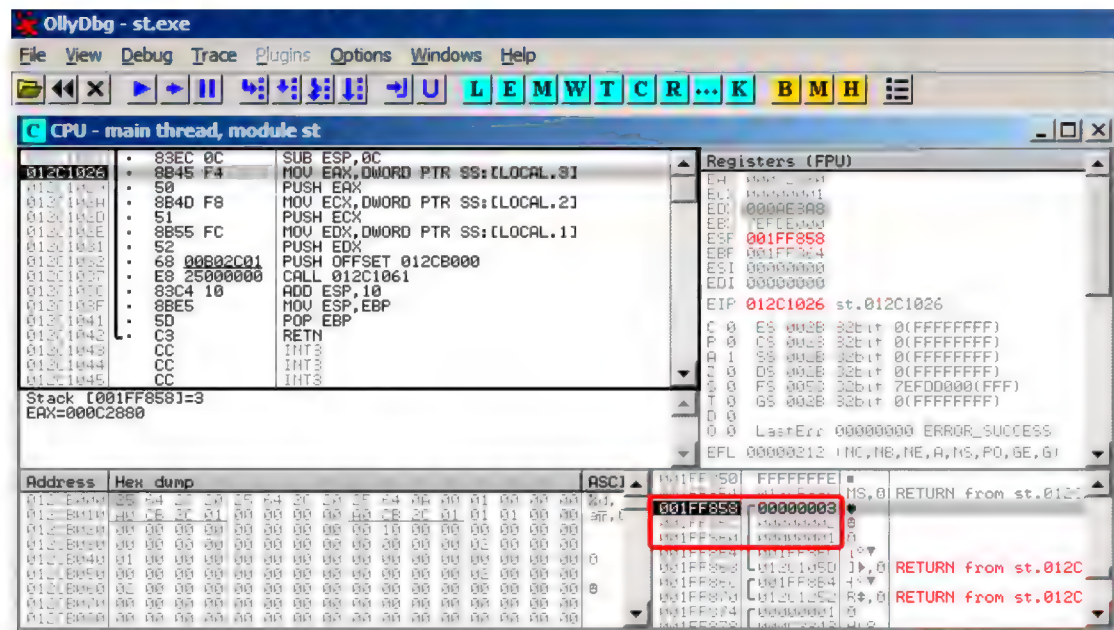


図 1.7: OllyDbg: f2()

... f2() の a, b, c は同じアドレスにあります！誰もまだ値を上書きしていないので、その時点でまだ変更はありません。したがって、この奇妙な状況が発生するためには、いくつかの関数を次々と呼び出さなければならず、**SP!**は各関数エントリで同じでなければならない（すなわち、それらは同じ数の引数を有する）。次に、ローカル変数はスタック内の同じ位置に配置されます。要約すると、スタック（およびメモリエル）内のすべての値は、以前の関数実行から残った値を持ちます。彼らは厳密な意味でランダムではなく、むしろ予測不可能な値を持っています。別のオプションがありますか？各関数の実行前にスタックの一部をクリアすることはおそらく可能ですが、余計な（そして不要な）作業です。

MSVC 2013

この例はMSVC 2010によってコンパイルされました。しかし、この本の読者は、このサンプルをMSVC 2013でコンパイルして実行し、3つの数字がすべて逆の結果になるでしょう。

```
c:\Polygon\c>st
3, 2, 1
```

どうして？私もMSVC 2013でこの例をコンパイルし、見てみました。

Listing 1.41: MSVC 2013

```
_a$ = -12      ; size = 4
_b$ = -8       ; size = 4
_c$ = -4       ; size = 4
```

```

_f2    PROC
...

_f2    ENDP

_c$ = -12      ; size = 4
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
_f1    PROC
...

_f1    ENDP

```

MSVC 2010とは異なり、MSVC 2013は関数 `f2()` の `a/b/c`変数を逆順に割り当てました。これは完全に正しい動作です。C/C++ 標準にはルールがありません。ローカル変数をローカルスタックに割り当てる必要があれば、どのような順番でもよいのです。理由の違いは、MSVC 2010にはその方法があり、MSVC 2013はおそらくコンパイラの心臓部で何かが変わったと考えられるからです。

第1.7.5節練習問題

- <http://challenges.re/51>
- <http://challenges.re/52>

第1.8節printf() 引数を取って

さて、ハローワールド! (1.5 on page 11) の例では、`main()` 関数本体の `printf()` を次のように置き換えます。

```

#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};

```

第1.8.1節x86

x86: 3つの引数

MSVC

MSVC 2010 Expressでコンパイルすると、

```

$SG3830 DB      'a=%d; b=%d; c=%d', 00H
...
        push     3
        push     2
        push     1
        push     OFFSET $SG3830
        call     _printf
        add      esp, 16                                ; 00000010H

```

ほぼ同じですが、`printf()` の引数が逆の順序でスタックにプッシュされるのが分かります。最初の引数は最後にプッシュされます。

ちなみに、32ビット環境での *int* 型の変数は、32ビット幅、つまり4バイトです。

だから、ここでは4つの引数があります。4 × 4 = 16。スタック内でちょうど16バイトを占める：文字列への32ビットポインタと *int* 型の3つの数字。

スタックポインタ (ESP レジスタ) が関数呼び出しの後の `ADD ESP, X` 命令によって元の状態に戻ったとき、関数引数の数はXを4で割るだけで推測できます。

もちろん、これは *cdecl* 呼び出し規約に固有のものであり、32ビット環境のみに適用されます。

呼び出し規約を参照してください。(?? on page ??)

いくつかの関数が互いに直後に戻る特定のケースでは、コンパイラは最後の呼び出しの後に複数の「`ADD ESP, X`」命令を1つにマージすることができます：

```

push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24

```

実際の例がここにあります。

Listing 1.42: x86

```

.text:100113E7  push     3
.text:100113E9  call     sub_100018B0 ; 引数を1つとる (3)
.text:100113EE  call     sub_100019D0 ; 引数をとらない
.text:100113F3  call     sub_10006A90 ; 引数をとらない
.text:100113F8  push     1
.text:100113FA  call     sub_100018B0 ; 引数を1つとる (1)
.text:100113FF  add      esp, 8      ; 一度にスタックから2つの引数を落とす

```

MSVC と OllyDbg

では、この例を OllyDbg に読み込みましょう。これは最もポピュラーなユーザーランド win32 デバッガーの1つです。MSVC 2012 で /MD オプションを使用してサンプルをコンパイルすることができます。これは MSVCRT*.DLL とリンクすることを意味し、インポートされた関数をデバッガではっきりと見ることができます。

その後、OllyDbg で実行可能ファイルをロードします。最初のブレークポイントは ntdll.dll にあり、F9 (実行) を押します。2番目のブレークポイントは CRT コードです。main() 関数を見つけなければなりません。

コードを一番上までスクロールしてコードを見つけます (MSVC はコードセクションの最初のところで main() 関数を割り当てます)。

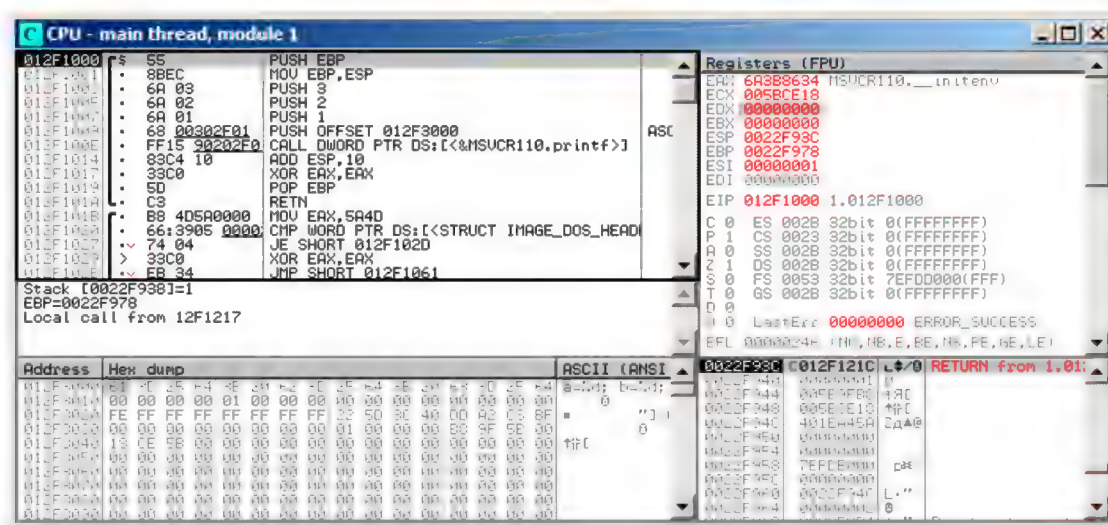


図 1.8: OllyDbg: the very start of the main() function

PUSH EBP 命令をクリックし、F2 (ブレークポイントを設定) を押し、F9 (実行) を押します。CRT コードをスキップするためには、これらの処理を実行する必要があります。なぜなら、実際にはまだ興味がないからです。

F8（ステップオーバー）を6回押します、つまり6つの命令をスキップします。

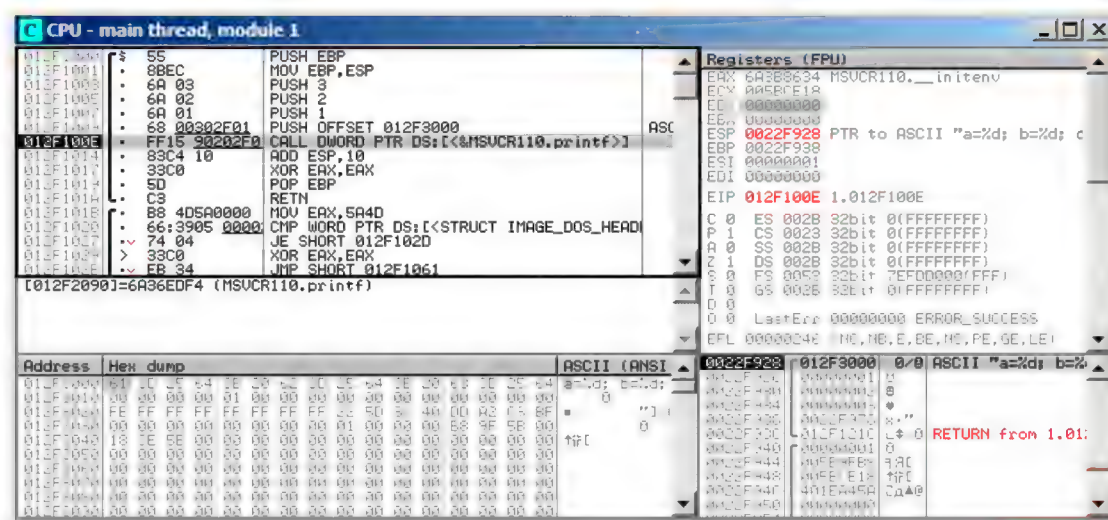


図 1.9: OllyDbg: before printf() execution

これで、**PC!**は CALL printf 命令を指し示します。OllyDbg は他のデバッガと同様に、変更されたレジスタの値を強調表示します。したがって、F8を押すたびに EIP が変化し、その値が赤で表示されます。引数の値がスタックにプッシュされるため、ESP も変更されます。

スタック内の値はどこにありますか？右下のデバッガーウィンドウを見てみましょう：

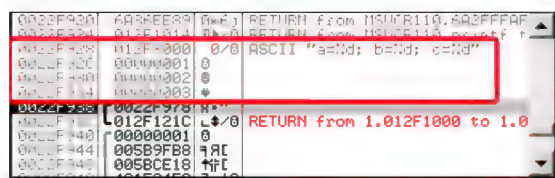


図 1.10: OllyDbg : 引数の値がプッシュされた後のスタック（赤い長方形の枠線はグラフィックエディタで作者によって追加されました）

スタック内のアドレス、スタック内の値、および追加の OllyDbg コメントが3つあります。OllyDbg は printf() のような文字列を理解しているので、ここに文字列とそれに付随する 3つの値を報告します。

フォーマット文字列を右クリックし、「Follow in dump」をクリックすると、フォーマット文字列がデバッガの左下のウィンドウに表示され、メモリの一部が常に表示されます。これらのメモリ値は編集できます。書式文字列を変更することができます。この場合、例の結果は異なります。この特殊なケースではそれほど有用ではありませんが、エクササイズとしてはいいかもしれません。

F8キーを押します（ステップオーバー）。

コンソールに次の出力が表示されます。

```
a=1; b=2; c=3
```

レジスタとスタックの状態がどのように変化したかを見てみましょう。

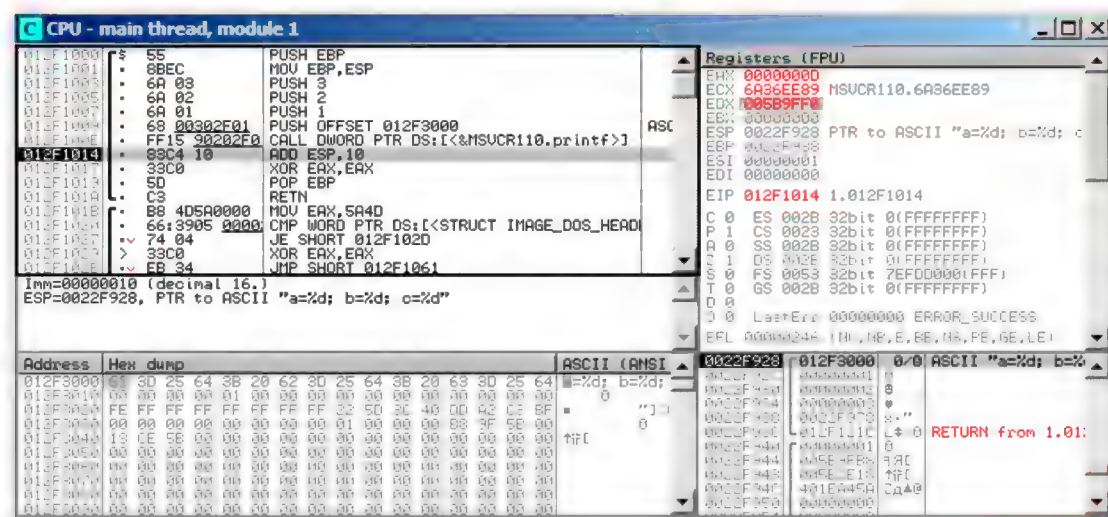


図 1.11: printf() 実行後の OllyDbg

レジスタ EAX に 0xD (13) が含まれるようになりました。printf() は印刷された文字数を返すので正しい。EIP の値は変更されました。実際、CALL printf の後に来る命令のアドレスを含んでいます。ECX と EDX の値も変更されています。明らかに、printf() 関数の隠れた機構は、自身の必要のため、それらを使用しました。

非常に重要な事実は、ESP 値もスタック状態も変更されていないことです！フォーマット文字列とそれに対応する3つの値がまだ存在することがわかります。実際には、これは cdecl 呼び出し規約の動作です：callee は ESP を以前の値に戻しません。caller はこれを行う責任があります。

F8をもう一度押して ADD ESP, 10 命令を実行します。

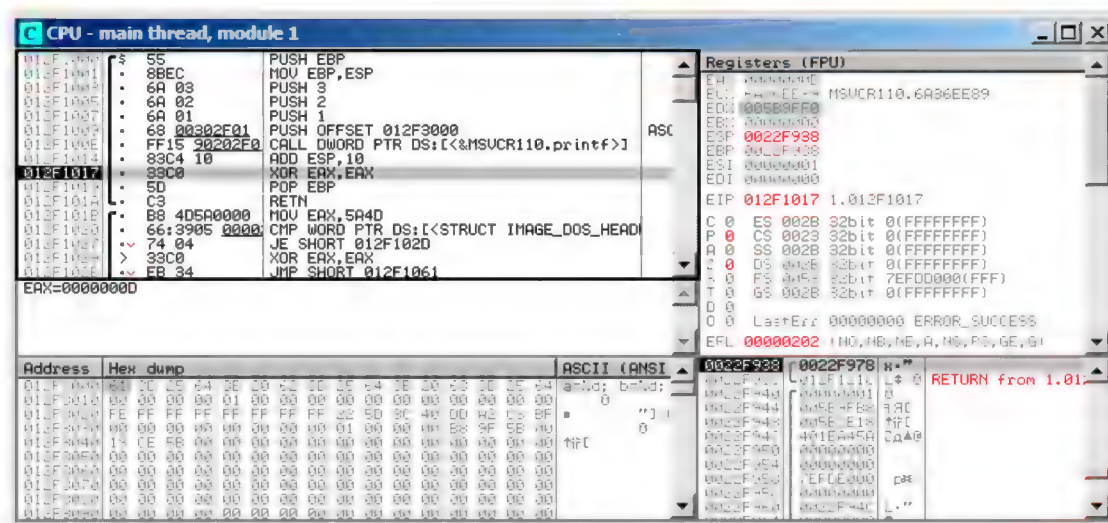


図 1.12: ADD ESP, 10 命令実行後の OllyDbg

ESP は変更されましたが、値はまだスタックにあります！はい、もちろん；これらの値をゼロなどに設定する必要はありません。スタックポインタ（**SP!**）の上にあるものはすべてノイズやガーベジであり、まったく意味がありません。とにかく、未使用のスタックエントリをクリアするのに時間がかかり、誰も本当に必要とはしません。

GCC

GCC 4.4.1を使って同じプログラムをLinuxでコンパイルして、IDA で何が得られたかを見ましょう。

main	proc near
var_10	= dword ptr -10h
var_C	= dword ptr -0Ch
var_8	= dword ptr -8
var_4	= dword ptr -4
	push ebp
	mov ebp, esp
	and esp, 0FFFFFFF0h
	sub esp, 10h
	mov eax, offset aADBD CD ; "a=%d; b=%d; c=%d"
	mov [esp+10h+var_4], 3
	mov [esp+10h+var_8], 2
	mov [esp+10h+var_C], 1
	mov [esp+10h+var_10], eax
	call _printf
	mov eax, 0
	leave

```
main          retn
              endp
```

MSVCコードとGCCコードの違いは、引数がスタックに格納される方法だけにあることに注目してください。ここでGCCは PUSH/POP を使用せずに、スタックを直接使用してスタックを操作しています

GCC and GDB

LinuxのGDB⁶⁶でもこの例を試してみましょう。

-g オプションは、デバッグ情報を実行可能ファイルに含めるようにコンパイラに指示します。

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.43: printf() にブレークポイントを設定しましょう

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

実行します。ここには printf() 関数のソースコードがありませんので、GDBはそれを表示することはできませんが、実行することはできます。

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29      printf.c: No such file or directory.
```

スタック要素を10個表示します。最も左の列には、スタック上のアドレスが含まれています。

```
(gdb) x/10w $esp
0xbffff11c:    0x0804844a    0x080484f0    0x00000001    0x00000002
0xbffff12c:    0x00000003    0x08048460    0x00000000    0x00000000
0xbffff13c:    0xb7e29905    0x00000001
```

最初の要素はRA (0x0804844a) です。このアドレスのメモリを逆アセンブルして確認することができます：

```
(gdb) x/5i 0x0804844a
0x804844a <main+45>: mov     $0x0,%eax
0x804844f <main+50>: leave
```

⁶⁶GNU Debugger

```
0x8048450 <main+51>: ret
0x8048451:   xchg   %ax,%ax
0x8048453:   xchg   %ax,%ax
```

2つの XCHG 命令は、NOPに類似したアイドル命令です。

2番目の要素 (0x080484f0) はフォーマット文字列アドレスです。

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

次の3つの要素 (1,2,3) は printf() の引数です。残りの要素はスタック上の「garbage」に過ぎないかもしれませんが、他の関数やローカル変数などからの値であってもかまいません。

「finish」を実行します。このコマンドは、GDBに「関数の最後まですべての命令を実行する」よう指示します。この場合、printf() の最後まで実行してください。

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at ↵
↳ printf.c:29
main () at 1.c:6
6          return 0;
Value returned is $2 = 13
```

GDBは、printf() が EAX (13) をリターンしたことを示しています。これは、OllyDbgの例のように、印刷される文字の数です。

また、「return 0;」と、この式が6行目の 1.c ファイルにあるという情報も表示されます。実際には、1.c ファイルは現在のディレクトリにあり、GDBはその文字列を見つけます。どのCコード行が現在実行されているかをGDBはどうやってに知るのでしょうか？これは、コンパイラがデバッグ情報を生成している間に、ソースコードの行番号と命令アドレスの間の関係のテーブルも保存しているためです。GDBはソースレベルのデバッガです。

レジスタを調べてみましょう。EAXには13が入っています

```
(gdb) info registers
eax          0xd          13
ecx          0x0          0
edx          0x0          0
ebx          0xb7fc0000    -1208221696
esp          0xbffffff120  0xbffffff120
ebp          0xbffffff138  0xbffffff138
esi          0x0          0
edi          0x0          0
eip          0x804844a      0x804844a <main+45>
...
```

現在の命令を逆アセンブルしましょう。矢印は、次に実行される命令を指し示します。

```
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>:   push   %ebp
0x0804841e <+1>:   mov    %esp,%ebp
```

```

0x08048420 <+3>:    and    $0xffffffff0,%esp
0x08048423 <+6>:    sub     $0x10,%esp
0x08048426 <+9>:    movl   $0x3,0xc(%esp)
0x0804842e <+17>:   movl   $0x2,0x8(%esp)
0x08048436 <+25>:   movl   $0x1,0x4(%esp)
0x0804843e <+33>:   movl   $0x80484f0, (%esp)
0x08048445 <+40>:   call  0x80482f0 <printf@plt>
=> 0x0804844a <+45>:   mov     $0x0,%eax
0x0804844f <+50>:   leave
0x08048450 <+51>:   ret
End of assembler dump.

```

GDBはデフォルトでAT&T構文を使用します。しかし、インテルの構文に切り替えることは可能です：

```

(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>:    push   ebp
0x0804841e <+1>:    mov    ebp,esp
0x08048420 <+3>:    and     esp,0xffffffff0
0x08048423 <+6>:    sub     esp,0x10
0x08048426 <+9>:    mov     DWORD PTR [esp+0xc],0x3
0x0804842e <+17>:   mov     DWORD PTR [esp+0x8],0x2
0x08048436 <+25>:   mov     DWORD PTR [esp+0x4],0x1
0x0804843e <+33>:   mov     DWORD PTR [esp],0x80484f0
0x08048445 <+40>:   call    0x80482f0 <printf@plt>
=> 0x0804844a <+45>:   mov     eax,0x0
0x0804844f <+50>:   leave
0x08048450 <+51>:   ret
End of assembler dump.

```

次の命令を実行します。**GDB**は、終わりの括弧を示します。つまり、ブロックを終了します。

```

(gdb) step
7      };

```

MOV EAX, 0 命令実行後のレジスタを調べてみましょう。実際、**EAX** はその時点で0です。

```

(gdb) info registers
eax            0x0            0
ecx            0x0            0
edx            0x0            0
ebx            0xb7fc0000      -1208221696
esp            0xbffff120      0xbffff120
ebp            0xbffff138      0xbffff138
esi            0x0            0
edi            0x0            0
eip            0x804844f        0x804844f <main+50>
...

```

x64: 8つの引数

他の引数がスタックを介してどのように渡されているかを知るために、引数の数を9 (printf() の書式文字列+ 8の int 変数) に増やして、この例を再度変更してみましょう。

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4,
    ↵ 4, 5, 6, 7, 8);
    return 0;
};
```

MSVC

前に述べたように、最初の4つの引数は、Win64の RCX、RDX、R8、R8 レジスタに渡されなければなりません。それはまさに私たちがここに見るものです。ただし、PUSH ではなく MOV 命令がスタックの準備に使用されるため、値は直接的にスタックに格納されます。

Listing 1.44: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H

main      PROC
sub       sub     rsp, 88

          mov     DWORD PTR [rsp+64], 8
          mov     DWORD PTR [rsp+56], 7
          mov     DWORD PTR [rsp+48], 6
          mov     DWORD PTR [rsp+40], 5
          mov     DWORD PTR [rsp+32], 4
          mov     r9d, 3
          mov     r8d, 2
          mov     edx, 1
          lea     rcx, OFFSET FLAT:$SG2923
          call    printf

          ; 0をリターン
          xor     eax, eax

          add     rsp, 88
          ret     0
main      ENDP
_TEXT    ENDS
END
```

注意深い読者は、4が十分であるときになぜ int 値のために8バイトが割り振られているのか尋ねるかもしれません。はい、思い出す必要があります: 64バイトより短い任意のデータ型に対して8バイトが割り当てられます。これは便宜上確立されています。任意の引数のアドレスを簡単に計算できます。また、それらはすべて整列したメモリアドレスに配置

されています。32ビット環境でも同じです。すべてのデータ型に4バイトが予約されています。

GCC

最初の6つの引数が RDI、RSI、RDX、RCX、R8、R9 レジスタを通過する点を除いて、画像はx86-64 *NIX OSの場合と似ています。残りすべてはスタックを介して。GCCは、RDI の代わりに EDI に文字列ポインタを格納するコードを生成しています。これまでは：[1.5.2 on page 20](#)

また、以前は printf() 呼び出しの前に EAX レジスタがクリアされていることに注意してください：[1.5.2 on page 20](#)

Listing 1.45: 最適化 GCC 4.4.6 x64

```
.LC0:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
    sub     rsp, 40

    mov     r9d, 5
    mov     r8d, 4
    mov     ecx, 3
    mov     edx, 2
    mov     esi, 1
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax ; ベクトルレジスタの数を渡す
    mov     DWORD PTR [rsp+16], 8
    mov     DWORD PTR [rsp+8], 7
    mov     DWORD PTR [rsp], 6
    call    printf

    ; 0をリターン

    xor     eax, eax
    add     rsp, 40
    ret
```

GCC + GDB

[GDB](#)でこの例を試してみましょう

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.
```


Listing 1.46: ブレークポイントを printf() に設定して実行しましょう

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d;
↳ ; g=%d; h=%d\n") at printf.c:29
29     printf.c: No such file or directory.
```

Registers RSI/RDX/RCX/R8/R9 have the expected values. RIP has the address of the very first instruction of the printf() function.

レジスタ RSI/RDX/RCX/R8/R9 は期待値を持っています。RIP には、printf() 関数の最初の命令のアドレスがあります。

```
(gdb) info registers
rax            0x0            0
rbx            0x0            0
rcx            0x3            3
rdx            0x2            2
rsi            0x1            1
rdi            0x400628 4195880
rbp            0x7fffffffdf60  0x7fffffffdf60
rsp            0x7fffffffdf38  0x7fffffffdf38
r8             0x4            4
r9             0x5            5
r10            0x7fffffffdfce0  140737488346336
r11            0x7ffff7a65f60  140737348263776
r12            0x400440 4195392
r13            0x7fffffffef040  140737488347200
r14            0x0            0
r15            0x0            0
rip            0x7ffff7a65f60  0x7ffff7a65f60 <__printf>
...
```

Listing 1.47: let's inspect the format string

```
(gdb) x/s $rdi
0x400628:      "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

今度はx/gコマンドでスタックをダンプしましょう。g は、*giant words*、つまり64ビットの単語を表します。

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x000000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007ffff7ffe048      0x0000000010000000
```

最初のスタック要素は、前の例と同様、RAです。3の値もスタックに通されます：6,7,8。また、クリアされていない32ビットの8が渡されたことがわかります：0x00007fff00000008。

値は *int* 型 (32ビット) なので、それは問題ありません。したがって、上位レジスタまたはスタック要素の部分には「ランダムなゴミ」が含まれている可能性があります。

`printf()` の実行後にコントロールが返る場所を見れば、**GDB**は `main()` 関数全体を表示します：

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x000000000400576
Dump of assembler code for function main:
   0x00000000040052d <+0>:    push    rbp
   0x00000000040052e <+1>:    mov     rbp, rsp
   0x000000000400531 <+4>:    sub     rsp, 0x20
   0x000000000400535 <+8>:    mov     DWORD PTR [rsp+0x10], 0x8
   0x00000000040053d <+16>:   mov     DWORD PTR [rsp+0x8], 0x7
   0x000000000400545 <+24>:   mov     DWORD PTR [rsp], 0x6
   0x00000000040054c <+31>:   mov     r9d, 0x5
   0x000000000400552 <+37>:   mov     r8d, 0x4
   0x000000000400558 <+43>:   mov     ecx, 0x3
   0x00000000040055d <+48>:   mov     edx, 0x2
   0x000000000400562 <+53>:   mov     esi, 0x1
   0x000000000400567 <+58>:   mov     edi, 0x400628
   0x00000000040056c <+63>:   mov     eax, 0x0
   0x000000000400571 <+68>:   call    0x400410 <printf@plt>
   0x000000000400576 <+73>:   mov     eax, 0x0
   0x00000000040057b <+78>:   leave
   0x00000000040057c <+79>:   ret
End of assembler dump.
```

`printf()` の実行を終了し、EAX をゼロにする命令を実行し、EAX レジスタが正確にゼロの値を持つことに注意してください。RIP は、LEAVE 命令、すなわち `main()` 関数の最後から2番目の命令を指すようになりました。

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=
↳ =%d; f=%d; g=%d; h=%d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6         return 0;
Value returned is $1 = 39
(gdb) next
7     };
(gdb) info registers
rax                0x0          0
rbx                0x0          0
rcx                0x26         38
rdx                0x7ffff7dd59f0 140737351866864
rsi                0x7fffffd9    2147483609
rdi                0x0          0
rbp                0x7ffffffffffd60 0x7ffffffffffd60
rsp                0x7ffffffffffd40 0x7ffffffffffd40
r8                 0x7ffff7dd26a0    140737351853728
r9                 0x7ffff7a60134    140737348239668
r10                0x7ffffffffffd5b0 140737488344496
r11                0x7ffff7a95900    140737348458752
```

```

r12      0x400440 4195392
r13      0x7fffffff040 140737488347200
r14      0x0      0
r15      0x0      0
rip      0x40057b 0x40057b <main+78>
...

```

第1.8.2節ARM

ARM: 3つの引数

引数を渡すためのARMの伝統的なスキーム（呼び出し規約）は、次のように動作します。最初の4つの引数は R0-R3 レジスタに渡されます。残りの引数はスタックを介して。これはfastcall (?? on page ??) またはwin64 (?? on page ??) の引数渡しスキームに似ています。

32ビットARM

非最適化 Keil 6/2013 (ARMモード)

Listing 1.48: 非最適化 Keil 6/2013 (ARMモード)

```

.text:00000000 main
.text:00000000 10 40 2D E9  STMFID  SP!, {R4,LR}
.text:00000004 03 30 A0 E3  MOV    R3, #3
.text:00000008 02 20 A0 E3  MOV    R2, #2
.text:0000000C 01 10 A0 E3  MOV    R1, #1
.text:00000010 08 00 8F E2  ADR    R0, aADBDCD      ; "a=%d; b=%d; c=%d"
.text:00000014 06 00 00 EB  BL     _2printf
.text:00000018 00 00 A0 E3  MOV    R0, #0          ; return 0
.text:0000001C 10 80 BD E8  LDMFD  SP!, {R4,PC}

```

したがって、最初の4つの引数は、R0-R3 レジスタをこの順序で渡します。R0 の printf() 形式文字列へのポインタ、R1 の1、R2 の2、R3 の3の順です。0x18 の命令は R0 に0を書き込みます。これは return 0 となるCの命令文です。珍しいことは何也没有什么ありません。

最適化 Keil 6/2013 は同じコードを生成します。

最適化 Keil 6/2013 (Thumbモード)

Listing 1.49: 最適化 Keil 6/2013 (Thumbモード)

```

.text:00000000 main
.text:00000000 10 B5      PUSH    {R4,LR}
.text:00000002 03 23      MOVNS  R3, #3
.text:00000004 02 22      MOVNS  R2, #2
.text:00000006 01 21      MOVNS  R1, #1
.text:00000008 02 A0      ADR     R0, aADBDCD      ; "a=%d; b=%d; c=%d"

```

```
.text:0000000A 00 F0 0D F8 BL      _2printf
.text:0000000E 00 20      MOV     R0, #0
.text:00000010 10 BD      POP     {R4,PC}
```

ARMモードの最適化されていないコードとの大きな違いはありません。

最適化 **Keil 6/2013 (ARMモード) + return**を削除してみる

`return 0` を取り除いて例を少し修正しましょう：

```
#include <stdio.h>

void main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
};
```

結果はちょっと珍しくなりました。

Listing 1.50: 最適化 Keil 6/2013 (ARMモード)

```
.text:00000014 main
.text:00000014 03 30 A0 E3 MOV     R3, #3
.text:00000018 02 20 A0 E3 MOV     R2, #2
.text:0000001C 01 10 A0 E3 MOV     R1, #1
.text:00000020 1E 0E 8F E2 ADR     R0, aADBDCD      ; "a=%d; b=%d; c=%d\n"
.text:00000024 CB 18 00 EA B      _2printf
```

これはARMモード用に最適化された (-O3) バージョンであり、今回は **B** を使い慣れた **BL** ではなく最後の命令と見なします。この最適化されたバージョンと前のバージョン（最適化なしでコンパイルされたもの）との別の違いは、関数のプロローグとエピローグ（**R0** と **LR**レジスタの値を保持する命令）の欠如です。**B** 命令は、x86の **JMP** と同様に、**LR**レジスタの操作なしで別のアドレスにジャンプするだけです。それはなぜ機能するのでしょうか？実際、このコードは以前のコードと事実上同等です。主な理由は2つあります。1) スタックも **SP!** (**スタックポインタ**) も変更されていません。2) `printf()` の呼び出しが最後の命令なので、その後は何も起こりません。完了すると、`printf()` 関数は **LR** に格納されているアドレスにコントロールを返します。**LR** は現在、関数が呼び出されたポイントのアドレスを格納しているので、`printf()` からの制御はそのポイントに返されます。したがって、**LR** を変更する必要がないため、**LR** を節約する必要はありません。`printf()` 以外の関数呼び出しがないため、**LR** を変更する必要はありません。さらに、この呼び出しの後、私たちは何もしません！これがそのような最適化が可能な理由です。

この最適化は、最後のステートメントが別の関数の呼び出しである関数でよく使用されます。同様の例をここに示します：[1.15.1 on page 191](#)

ARM64

Listing 1.51: 非最適化 GCC (Linaro) 4.9

```

.LC1:
    .string "a=%d; b=%d; c=%d"
f2:
; スタックフレームにFPとLRを保存する:
    stp    x29, x30, [sp, -16]!
; スタックフレームを設定する (FP=SP):
    add    x29, sp, 0
    adrp   x0, .LC1
    add    x0, x0, :lo12:.LC1
    mov    w1, 1
    mov    w2, 2
    mov    w3, 3
    bl     printf
    mov    w0, 0
; FP と LRをリストアする
    ldp    x29, x30, [sp], 16
    ret

```

第1の命令 STP (ストアペア) は、FP (X29) およびLR (X30) をスタックに保存します。2番目の ADD X29, SP, 0 命令がスタックフレームを形成します。SP!の値をX29に書き込むだけです。

次に、使い慣れた ADRP/ADD 命令ペアを参照します。これは、文字列へのポインタを形成します。すなわち、lo12 は、LC1アドレスの下位12ビットを ADD 命令のオペコードに書き込みます。printf() の文字列書式の%d は32ビット int なので、1,2および3は32ビットのレジスタ部分にロードされます。

最適化 GCC (Linaro) 4.9 は同じコードを生成します。

ARM: 8つの引数

前のセクションの9つの引数を使って例を再利用してみましょう：[1.8.1 on page 63](#)

```

#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4,
    ↪ 4, 5, 6, 7, 8);
    return 0;
};

```

最適化 Keil 6/2013: ARMモード

```

.text:00000028          main
.text:00000028

```

```

.text:00000028          var_18 = -0x18
.text:00000028          var_14 = -0x14
.text:00000028          var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5 STR    LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2 SUB    SP, SP, #0x14
.text:00000030 08 30 A0 E3 MOV    R3, #8
.text:00000034 07 20 A0 E3 MOV    R2, #7
.text:00000038 06 10 A0 E3 MOV    R1, #6
.text:0000003C 05 00 A0 E3 MOV    R0, #5
.text:00000040 04 C0 8D E2 ADD    R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8 STMIA  R12, {R0-R3}
.text:00000048 04 00 A0 E3 MOV    R0, #4
.text:0000004C 00 00 8D E5 STR    R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3 MOV    R3, #3
.text:00000054 02 20 A0 E3 MOV    R2, #2
.text:00000058 01 10 A0 E3 MOV    R1, #1
.text:0000005C 6E 0F 8F E2 ADR    R0, aADBDCCDDDEDFDGD ; "a=%d; b=%d; c=%d;
    d=%d; e=%d; f=%d; g=%"...
.text:00000060 BC 18 00 EB BL     __2printf
.text:00000064 14 D0 8D E2 ADD    SP, SP, #0x14
.text:00000068 04 F0 9D E4 LDR    PC, [SP+4+var_4],#4

```

このコードはいくつかの部分に分けることができます：

- 関数プロローグ:

最初の STR LR, [SP,#var_4]! 命令は、このレジスタを printf() 呼び出しに使用する予定であるため、**LR**をスタックに保存します。最後の感嘆符は、事前索引を示します。

これは、まず**SP!**を4減少させた後、**SP!**に格納されたアドレスに**LR**を保存することを意味します。これはx86のPUSHに似ています。もっと読む：[1.30.2 on page 536](#)

第2の SUB SP, SP, #0x14 命令は、スタック上に0x14 (20) バイトを割り当てるために**SP!** (**スタックポインタ**) を減少させる。実際には、スタックを介して5つの32ビット値を printf() 関数に渡さなければならず、それぞれが4バイトを占めます。これは正確に $5 * 4 = 20$ です。他の4つの32ビット値は、レジスタに通される。

- スタックを介して5,6,7および8を渡す：それらはそれぞれ R0, R1, R2 と R3 レジスタに格納される。

次に、ADD R12, SP, #0x18+var_14 命令は、これら4つの変数が格納されるスタックアドレスを R12 レジスタに書き込みます。var_14 は、スタックにアクセスするコードを便利に表示するために **IDA** によって作成された-0x14に等しいアセンブリマクロです。**IDA** によって生成される var_? マクロは、スタック内のローカル変数を反映します。

したがって、SP+4 は R12 レジスタに格納されます。

次の STMIA R12, R0-R3 命令は、レジスタ R0-R3 の内容を R12 が指すメモリに書き込みます。STMIA は、後に複数のインクリメントを格納します。*Increment After* は、各レジスタ値が書き込まれた後に R12 が4ずつ増加することを意味する。

- スタックを介して4を渡す：4が R0 に格納され、STR R0, [SP,#0x18+var_18] 命令の助けを借りてこの値がスタックに保存されます。var_18 は-0x18なので、オフ

セットは0になります。したがって、R0 レジスタ (4) の値は**SP!**に書き込まれたアドレスに書き込まれます。

- レジスタ経由で1,2,3を渡す最初の3つの数値 (a、b、c) (それぞれ1,2,3) の値は、printf () 呼び出しの直前に R1、R2、R3 レジスタに渡されます。
- printf() 呼び出し。
- 関数エピローグ：

ADD SP, SP, #0x14 命令は**SP!**ポインタを元の値に戻して、スタックに格納されているものをすべて取り消します。もちろん、スタックに格納されているものはそこにとどまりますが、後続の関数の実行中にすべて書き換えられます。

LDR PC, [SP+4+var_4], #4 命令は、保存された**LR**値をスタックから**PC!**レジスタにロードして、機能を終了させます。感嘆符はありません。次に、**SP!**は $(4 + var_4 = 4 + (-4) = 0)$ に格納されているアドレスから最初にロードされるため、この命令は LDR PC, [SP], #4 に似ています)、**SP!**は4だけ増加します。これはポストインデックス⁶⁷と呼ばれます。なぜ **IDA** はそのような指示を表示するのですか？なぜなら、スタックレイアウトと、var_4 がローカルスタックの**LR**値を保存するために割り当てられているという事実を説明したいからです。この命令は、x86の POP PC と多少似ています。⁶⁸

最適化 Keil 6/2013: Thumbモード

```
.text:0000001C      printf_main2
.text:0000001C
.text:0000001C      var_18 = -0x18
.text:0000001C      var_14 = -0x14
.text:0000001C      var_8  = -8
.text:0000001C
.text:0000001C 00 B5      PUSH    {LR}
.text:0000001E 08 23      MOVS    R3, #8
.text:00000020 85 B0      SUB     SP, SP, #0x14
.text:00000022 04 93      STR     R3, [SP, #0x18+var_8]
.text:00000024 07 22      MOVS    R2, #7
.text:00000026 06 21      MOVS    R1, #6
.text:00000028 05 20      MOVS    R0, #5
.text:0000002A 01 AB      ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3      STMIA   R3!, {R0-R2}
.text:0000002E 04 20      MOVS    R0, #4
.text:00000030 00 90      STR     R0, [SP, #0x18+var_18]
.text:00000032 03 23      MOVS    R3, #3
.text:00000034 02 22      MOVS    R2, #2
.text:00000036 01 21      MOVS    R1, #1
.text:00000038 A0 A0      ADR     R0, aADBDCDDDED FDGD ; "a=%d; b=%d; c=%d;
      d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8 BL      __2printf
.text:0000003E
.text:0000003E      loc_3E ; CODE XREF: example13_f+16
```

⁶⁷ もっと読む：1.30.2 on page 536

⁶⁸ x86では POP を使って IP/EIP/RIP の値を設定することは不可能ですが、アナロジーとしてはよいでしょう

```
.text:0000003E 05 B0      ADD     SP, SP, #0x14
.text:00000040 00 BD      POP     {PC}
```

出力は前の例とほぼ同じです。ただし、これはThumbコードであり、値はスタックに別々にパックされます。8が先に進み、次に5,6,7、および4が3番目に進みます。

最適化 Xcode 4.6.3 (LLVM): ARMモード

```
__text:0000290C      _printf_main2
__text:0000290C
__text:0000290C      var_1C = -0x1C
__text:0000290C      var_C  = -0xC
__text:0000290C
__text:0000290C 80 40 2D E9      STMFD   SP!, {R7,LR}
__text:00002910 0D 70 A0 E1      MOV     R7, SP
__text:00002914 14 D0 4D E2      SUB     SP, SP, #0x14
__text:00002918 70 05 01 E3      MOV     R0, #0x1570
__text:0000291C 07 C0 A0 E3      MOV     R12, #7
__text:00002920 00 00 40 E3      MOVT    R0, #0
__text:00002924 04 20 A0 E3      MOV     R2, #4
__text:00002928 00 00 8F E0      ADD     R0, PC, R0
__text:0000292C 06 30 A0 E3      MOV     R3, #6
__text:00002930 05 10 A0 E3      MOV     R1, #5
__text:00002934 00 20 8D E5      STR     R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9      STMFA   SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3      MOV     R9, #8
__text:00002940 01 10 A0 E3      MOV     R1, #1
__text:00002944 02 20 A0 E3      MOV     R2, #2
__text:00002948 03 30 A0 E3      MOV     R3, #3
__text:0000294C 10 90 8D E5      STR     R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB      BL      _printf
__text:00002954 07 D0 A0 E1      MOV     SP, R7
__text:00002958 80 80 BD E8      LDMFD   SP!, {R7,PC}
```

STMFA (Store Multiple Increment Before) 命令の同義語である STMIB (Store Multiple Full Ascending) 命令を除き、既に見てきたものとほぼ同じです。この命令は、**SP!**レジスタの値を増加させ、逆の順序でこれら2つの動作を実行するのではなく、次のレジスタ値をメモリに書き込むだけです。

目を引くもう一つのこと、命令が一見無作為に配置されていることです。例えば、R0レジスタの値は、アドレス 0x2918, 0x2920 and 0x2928 の3箇所で操作できます。

しかしながら、最適化コンパイラは、実行中により高い効率を達成するために、命令をどのように順序付けするかに関する独自の理由を有することができる。

通常、プロセッサは、並んで配置された命令を同時に実行しようと試みます。たとえば、MOVT R0, #0、ADD R0, PC, R0 などの命令は、両方とも R0 レジスタを変更するため、同時に実行することはできません。一方、MOVT R0, #0、MOV R2, #4 命令は、実行の影響が互いに矛盾しないため、同時に実行することができます。おそらく、コンパイラはそのような方法でコードを生成しようと試みます (どこでも可能です)。

最適化 Xcode 4.6.3 (LLVM): Thumb-2モード

```

__text:00002BA0          _printf_main2
__text:00002BA0
__text:00002BA0          var_1C = -0x1C
__text:00002BA0          var_18 = -0x18
__text:00002BA0          var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5          PUSH     {R7,LR}
__text:00002BA2 6F 46          MOV      R7, SP
__text:00002BA4 85 B0          SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20     MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C     MOV.W    R12, #7
__text:00002BAE C0 F2 00 00     MOVT.W   R0, #0
__text:00002BB2 04 22          MOVS     R2, #4
__text:00002BB4 78 44          ADD      R0, PC    ; char *
__text:00002BB6 06 23          MOVS     R3, #6
__text:00002BB8 05 21          MOVS     R1, #5
__text:00002BBA 0D F1 04 0E     ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92          STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09     MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10     STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21          MOVS     R1, #1
__text:00002BCA 02 22          MOVS     R2, #2
__text:00002BCC 03 23          MOVS     R3, #3
__text:00002BCE CD F8 10 90     STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA     BLX      _printf
__text:00002BD6 05 B0          ADD      SP, SP, #0x14
__text:00002BD8 80 BD          POP      {R7,PC}

```

Thumb/Thumb-2命令が代わりに使用される点を除いて、出力は前の例とほぼ同じです。

ARM64

非最適化 GCC (Linaro) 4.9

Listing 1.52: 非最適化 GCC (Linaro) 4.9

```

.LC2:
    .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
f3:
; スタックにスペースをあける:
    sub     sp, sp, #32
; スタックフレームにFPとLRを保存する:
    stp     x29, x30, [sp,16]
; スタックフレームを設定する (FP=SP+16):
    add     x29, sp, 16
    adrp    x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
    add     x0, x0, :lo12:.LC2

```

```

        mov     w1, 8           ; 9th argument
        str     w1, [sp]       ; store 9th argument in the stack
        mov     w1, 1
        mov     w2, 2
        mov     w3, 3
        mov     w4, 4
        mov     w5, 5
        mov     w6, 6
        mov     w7, 7
        bl      printf
        sub     sp, x29, #16
; FPとLRとリストアする
        ldp     x29, x30, [sp,16]
        add     sp, sp, 32
        ret

```

最初の8つの引数は、XレジスタまたはWレジスタに渡されます。[*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]⁶⁹。文字列ポインタは64ビットのレジスタを必要とするため、X0で渡されます。それ以外の値はすべてint型32ビット型なので、レジスタ (W-) の32ビット部分に格納されます。第9引数 (8) はスタックを介して渡されます。実際には、レジスタの数が限られているため、多数の引数をレジスタに渡すことはできません。

最適化 GCC (Linaro) 4.9 は同じコードを生成します。

第1.8.3節MIPS

3 arguments

最適化 GCC 4.4.5

「ハローワールド!」の例との主な違いは、puts() の代わりに printf() が呼び出され、さらに3つの引数がレジスタ \$5...\$7 (または \$A0...\$A2) に渡されるという点です。そのため、これらのレジスタの前にAが付いています。これは、関数引数の受け渡しに使用されることを意味します。

Listing 1.53: 最適化 GCC 4.4.5 (アセンブリ出力)

```

$LC0:
        .ascii  "a=%d; b=%d; c=%d\000"
main:
; 関数プロローグ:
        lui     $28,%hi(__gnu_local_gp)
        addiu   $sp,$sp,-32
        addiu   $28,$28,%lo(__gnu_local_gp)
        sw      $31,28($sp)
; printf() のアドレスをロードする:
        lw      $25,%call16(printf)($28)
; テキスト文字列のアドレスをロードし、printf() の1番目の引数を設定する:

```

⁶⁹以下で利用可能 http://infocenter.arm.com/help/topic/com.arm.doc.ih10055b/IH10055B_aapcs64.pdf

```

        lui    $4,%hi($LC0)
        addiu   $4,$4,%lo($LC0)
; printf() の2番目の引数を設定する:
        li     $5,1                # 0x1
; printf() の3番目の引数を設定する:
        li     $6,2                # 0x2
; printf() をコールする:
        jalr    $25
; printf() の4番目の引数を設定する (分岐遅延スロット):
        li     $7,3                # 0x3

; 関数エピローグ:
        lw     $31,28($sp)
; 戻り値に0を設定する:
        move    $2,$0
; リターン
        j       $31
        addiu   $sp,$sp,32 ; branch delay slot

```

Listing 1.54: 最適化 GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_10             = -0x10
.text:00000000 var_4              = -4
.text:00000000
; 関数プロローグ:
.text:00000000                lui    $gp, (__gnu_local_gp >> 16)
.text:00000004                addiu   $sp, -0x20
.text:00000008                la     $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C                sw     $ra, 0x20+var_4($sp)
.text:00000010                sw     $gp, 0x20+var_10($sp)
; printf() のアドレスをロードする:
.text:00000014                lw     $t9, (printf & 0xFFFF)($gp)
; テキスト文字列のアドレスをロードし、printf() の1番目の引数を設定する:
.text:00000018                la     $a0, $LC0                # "a=%d; b=%d; c=%d"
; printf() の2番目の引数を設定する:
.text:00000020                li     $a1, 1
; printf() の3番目の引数を設定する:
.text:00000024                li     $a2, 2
; printf() をコールする:
.text:00000028                jalr    $t9
; printf() の4番目の引数を設定する (分岐遅延スロット):
.text:0000002C                li     $a3, 3
; 関数エピローグ:
.text:00000030                lw     $ra, 0x20+var_4($sp)
; 戻り値に0を設定する:
.text:00000034                move    $v0, $zero
; リターン
.text:00000038                jr     $ra
.text:0000003C                addiu   $sp, 0x20 ; branch delay slot

```

IDA は、LUI および ADDIU 命令のペアを1つの LA 疑似命令に統合しました。だから、ア

ドレス0x1Cに命令がないのは、LA が8バイトを占有しているからです。

非最適化 GCC 4.4.5

非最適化 GCC はもっと冗長です。

Listing 1.55: 非最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
    .ascii  "a=%d; b=%d; c=%d\000"
main:
; 関数プロローグ:
    addiu   $sp,$sp,-32
    sw      $31,28($sp)
    sw      $fp,24($sp)
    move    $fp,$sp
    lui     $28,%hi(__gnu_local_gp)
    addiu   $28,$28,%lo(__gnu_local_gp)
; テキスト文字列のアドレスをロードする:
    lui     $2,%hi($LC0)
    addiu   $2,$2,%lo($LC0)
; printf() の1番目の引数を設定する:
    move    $4,$2
; printf() の2番目の引数を設定する:
    li      $5,1                # 0x1
; printf() の3番目の引数を設定する:
    li      $6,2                # 0x2
; printf() の4番目の引数を設定する:
    li      $7,3                # 0x3
; printf() のアドレスを取得する:
    lw      $2,%call16(printf)($28)
    nop
; printf() をコールする:
    move    $25,$2
    jalr    $25
    nop

; 関数エピローグ:
    lw      $28,16($fp)
; 戻り値に0を設定する:
    move    $2,$0
    move    $sp,$fp
    lw      $31,28($sp)
    lw      $fp,24($sp)
    addiu   $sp,$sp,32
; リターン
    j       $31
    nop
```

Listing 1.56: 非最適化 GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
```



```

.text:00000000 var_10      = -0x10
.text:00000000 var_8      = -8
.text:00000000 var_4      = -4
.text:00000000
; 関数プロローグ:
.text:00000000          addiu   $sp, -0x20
.text:00000004          sw      $ra, 0x20+var_4($sp)
.text:00000008          sw      $fp, 0x20+var_8($sp)
.text:0000000C          move    $fp, $sp
.text:00000010          la      $gp, __gnu_local_gp
.text:00000018          sw      $gp, 0x20+var_10($sp)
; テキスト文字列のアドレスをロード:
.text:0000001C          la      $v0, aADBDCD      # "a=%d; b=%d; c=%d"
; printf() の1番目の引数を設定:
.text:00000024          move    $a0, $v0
; printf() の2番目の引数を設定:
.text:00000028          li      $a1, 1
; printf() の3番目の引数を設定:
.text:0000002C          li      $a2, 2
; printf() の4番目の引数を設定:
.text:00000030          li      $a3, 3
; printf() のアドレスを得る:
.text:00000034          lw      $v0, (printf & 0xFFFF)($gp)
.text:00000038          or      $at, $zero
; printf() をコールする:
.text:0000003C          move    $t9, $v0
.text:00000040          jalr    $t9
.text:00000044          or      $at, $zero ; NOP
; 関数エピローグ:
.text:00000048          lw      $gp, 0x20+var_10($fp)
; 戻り値に0を設定する:
.text:0000004C          move    $v0, $zero
.text:00000050          move    $sp, $fp
.text:00000054          lw      $ra, 0x20+var_4($sp)
.text:00000058          lw      $fp, 0x20+var_8($sp)
.text:0000005C          addiu   $sp, 0x20
; リターン
.text:00000060          jr      $ra
.text:00000064          or      $at, $zero ; NOP

```

8つの引数

前のセクションの9つの引数を使用して、例を再度使用してみましょう：[1.8.1 on page 63](#)

```

#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4,
    ↪ 4, 5, 6, 7, 8);
    return 0;
};

```

最適化 GCC 4.4.5

最初の4つの引数だけが \$A0 ...\$A3 レジスタに渡され、残りはスタックを介して渡されます。

これはO32呼び出し規約 (MIPS世界で最も一般的なもの) です。他の呼び出し規則 (N32のような) は、異なる目的のためにレジスタを使用するかもしれません。

SW は「Store Word」(レジスタからメモリへ) の略語です。MIPSには値をメモリに格納する命令がないため、代わりに命令ペア (LI/SW) を使用する必要があります。

Listing 1.57: 最適化 GCC 4.4.5 (アセンブリ出力)

```
$LC0:
    .ascii "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; 関数プロローグ:
    lui    $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-56
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw      $31,52($sp)
; スタックに5番目の引数を渡す:
    li      $2,4                # 0x4
    sw      $2,16($sp)
; スタックに6番目の引数を渡す:
    li      $2,5                # 0x5
    sw      $2,20($sp)
; スタックに7番目の引数を渡す:
    li      $2,6                # 0x6
    sw      $2,24($sp)
; スタックに8番目の引数を渡す:
    li      $2,7                # 0x7
    lw      $25,%call16(sprintf)($28)
    sw      $2,28($sp)
; $a0に1番目の引数を渡す:
    lui     $4,%hi($LC0)
; スタックに9番目の引数を渡す:
    li      $2,8                # 0x8
    sw      $2,32($sp)
    addiu   $4,$4,%lo($LC0)
; $a1に2番目の引数を渡す:
    li      $5,1                # 0x1
; $a2に3番目の引数を渡す:
    li      $6,2                # 0x2
; printf() をコールする:
    jalr    $25
; $a3に4番目の引数を渡す (分岐遅延スロット):
    li      $7,3                # 0x3

; 関数エピローグ:
    lw      $31,52($sp)
; 戻り値に0を設定する:
    move    $2,$0
; リターン
```

```

j      $31
addiu  $sp,$sp,56 ; branch delay slot

```

Listing 1.58: 最適化 GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_28      = -0x28
.text:00000000 var_24      = -0x24
.text:00000000 var_20      = -0x20
.text:00000000 var_1C      = -0x1C
.text:00000000 var_18      = -0x18
.text:00000000 var_10      = -0x10
.text:00000000 var_4       = -4
.text:00000000
; 関数プロローグ:
.text:00000000          lui    $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu   $sp, -0x38
.text:00000008          la     $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw     $ra, 0x38+var_4($sp)
.text:00000010          sw     $gp, 0x38+var_10($sp)
; スタックに5番目の引数を渡す:
.text:00000014          li     $v0, 4
.text:00000018          sw     $v0, 0x38+var_28($sp)
; スタックに6番目の引数を渡す:
.text:0000001C          li     $v0, 5
.text:00000020          sw     $v0, 0x38+var_24($sp)
; スタックに7番目の引数を渡す:
.text:00000024          li     $v0, 6
.text:00000028          sw     $v0, 0x38+var_20($sp)
; スタックに8番目の引数を渡す:
.text:0000002C          li     $v0, 7
.text:00000030          lw     $t9, (printf & 0xFFFF)($gp)
.text:00000034          sw     $v0, 0x38+var_1C($sp)
; $a0に1番目の引数を準備する:
.text:00000038          lui    $a0, ($LC0 >> 16) # "a=%d; b=%d;
          c=%d; d=%d; e=%d; f=%d; g=%"...
; スタックに9番目の引数を渡す:
.text:0000003C          li     $v0, 8
.text:00000040          sw     $v0, 0x38+var_18($sp)
; $a0に1番目の引数を渡す:
.text:00000044          la     $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d;
          c=%d; d=%d; e=%d; f=%d; g=%"...
; $a1に2番目の引数を渡す:
.text:00000048          li     $a1, 1
; $a2に3番目の引数を渡す:
.text:0000004C          li     $a2, 2
; printf() をコールする:
.text:00000050          jalr   $t9
; $a3に4番目の引数を渡すpass 4th argument in $a3 (分岐遅延スロット):
.text:00000054          li     $a3, 3
; 関数エピローグ:
.text:00000058          lw     $ra, 0x38+var_4($sp)
; 戻り値に0を設定する:

```

.text:0000005C	move	\$v0, \$zero
; リターン		
.text:00000060	jr	\$ra
.text:00000064	addiu	\$sp, 0x38 ; branch delay slot

非最適化 GCC 4.4.5

非最適化 GCC はもっと冗長です。

Listing 1.59: 非最適化 GCC 4.4.5 (アセンブリ出力)

```

$LC0:
    .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"
main:
; 関数プロローグ:
    addiu    $sp,$sp,-56
    sw       $31,52($sp)
    sw       $fp,48($sp)
    move     $fp,$sp
    lui      $28,%hi(__gnu_local_gp)
    addiu    $28,$28,%lo(__gnu_local_gp)
    lui      $2,%hi($LC0)
    addiu    $2,$2,%lo($LC0)
; スタックに5番目の引数を渡す:
    li       $3,4                # 0x4
    sw       $3,16($sp)
; スタックに6番目の引数を渡す:
    li       $3,5                # 0x5
    sw       $3,20($sp)
; スタックに7番目の引数を渡す:
    li       $3,6                # 0x6
    sw       $3,24($sp)
; スタックに8番目の引数を渡す:
    li       $3,7                # 0x7
    sw       $3,28($sp)
; スタックに9番目の引数を渡す:
    li       $3,8                # 0x8
    sw       $3,32($sp)
; $a0に1番目の引数を渡す:
    move     $4,$2
; $a1に2番目の引数を渡す:
    li       $5,1                # 0x1
; $a2に3番目の引数を渡す:
    li       $6,2                # 0x2
; $a3に4番目の引数を渡す:
    li       $7,3                # 0x3
; printf() をコールする:
    lw       $2,%call16(printf)($28)
    nop
    move     $25,$2
    jalr     $25
    nop

```

```

; 関数エピローグ:
    lw      $28,40($fp)
; 戻り値に0を設定する:
    move    $2,$0
    move    $sp,$fp
    lw      $31,52($sp)
    lw      $fp,48($sp)
    addiu   $sp,$sp,56
; リターン
    j       $31
    nop

```

Listing 1.60: 非最適化 GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_28          = -0x28
.text:00000000 var_24          = -0x24
.text:00000000 var_20          = -0x20
.text:00000000 var_1C          = -0x1C
.text:00000000 var_18          = -0x18
.text:00000000 var_10          = -0x10
.text:00000000 var_8           = -8
.text:00000000 var_4           = -4
.text:00000000
; 関数プロローグ:
.text:00000000                addiu   $sp, -0x38
.text:00000004                sw      $ra, 0x38+var_4($sp)
.text:00000008                sw      $fp, 0x38+var_8($sp)
.text:0000000C                move    $fp, $sp
.text:00000010                la      $gp, __gnu_local_gp
.text:00000018                sw      $gp, 0x38+var_10($sp)
.text:0000001C                la      $v0, aADBDCDDDEDFDGD # "a=%d; b=%d;
                        c=%d; d=%d; e=%d; f=%d; g=%"...
; スタックに5番目の引数を渡す:
.text:00000024                li      $v1, 4
.text:00000028                sw      $v1, 0x38+var_28($sp)
; スタックに6番目の引数を渡す:
.text:0000002C                li      $v1, 5
.text:00000030                sw      $v1, 0x38+var_24($sp)
; スタックに7番目の引数を渡す:
.text:00000034                li      $v1, 6
.text:00000038                sw      $v1, 0x38+var_20($sp)
; スタックに8番目の引数を渡す:
.text:0000003C                li      $v1, 7
.text:00000040                sw      $v1, 0x38+var_1C($sp)
; スタックに9番目の引数を渡す:
.text:00000044                li      $v1, 8
.text:00000048                sw      $v1, 0x38+var_18($sp)
; $a0に1番目の引数を渡す:
.text:0000004C                move    $a0, $v0
; $a1に2番目の引数を渡す:
.text:00000050                li      $a1, 1
; $a2に3番目の引数を渡す:

```

```

.text:00000054      li      $a2, 2
; $a3に4番目の引数を渡す:
.text:00000058      li      $a3, 3
; printf() をコールする:
.text:0000005C      lw      $v0, (printf & 0xFFFF)($gp)
.text:00000060      or      $at, $zero
.text:00000064      move    $t9, $v0
.text:00000068      jalr    $t9
.text:0000006C      or      $at, $zero ; NOP
; 関数エピローグ:
.text:00000070      lw      $gp, 0x38+var_10($fp)
; 戻り値に0を設定する:
.text:00000074      move    $v0, $zero
.text:00000078      move    $sp, $fp
.text:0000007C      lw      $ra, 0x38+var_4($sp)
.text:00000080      lw      $fp, 0x38+var_8($sp)
.text:00000084      addiu   $sp, 0x38
; リターン
.text:00000088      jr      $ra
.text:0000008C      or      $at, $zero ; NOP

```

第1.8.4節結論

関数呼び出しの概略を以下に示します。

Listing 1.61: x86

```

...
PUSH 3rd argument
PUSH 2nd argument
PUSH 1st argument
CALL function
; スタックポインタを修正する (必要なら)

```

Listing 1.62: x64 (MSVC)

```

MOV RCX, 1st argument
MOV RDX, 2nd argument
MOV R8, 3rd argument
MOV R9, 4th argument
...
PUSH 5th, 6th argument, etc. (if needed)
CALL function
; スタックポインタを修正する (必要なら)

```

Listing 1.63: x64 (GCC)

```

MOV RDI, 1st argument
MOV RSI, 2nd argument
MOV RDX, 3rd argument
MOV RCX, 4th argument
MOV R8, 5th argument
MOV R9, 6th argument

```



```
...
PUSH 7th, 8th argument, etc. (if needed)
CALL function
; スタックポインタを修正する (必要なら)
```

Listing 1.64: ARM

```
MOV R0, 1st argument
MOV R1, 2nd argument
MOV R2, 3rd argument
MOV R3, 4th argument
; 5番目、6番目の引数などをスタックに渡す (必要なら)
BL function
; スタックポインタを修正する (必要なら)
```

Listing 1.65: ARM64

```
MOV X0, 1st argument
MOV X1, 2nd argument
MOV X2, 3rd argument
MOV X3, 4th argument
MOV X4, 5th argument
MOV X5, 6th argument
MOV X6, 7th argument
MOV X7, 8th argument
; 9番目、10番目の引数などをスタックに渡す (必要なら)
BL function
; スタックポインタを修正する (必要なら)
```

Listing 1.66: MIPS (O32 calling convention)

```
LI $4, 1st argument ; AKA $A0
LI $5, 2nd argument ; AKA $A1
LI $6, 3rd argument ; AKA $A2
LI $7, 4th argument ; AKA $A3
; 5番目、6番目の引数などをスタックに渡す (必要なら)
LW temp_reg, address of function
JALR temp_reg
```

第1.8.5節ところで

ところで、x86、x64、fastcall、ARM、およびMIPSで渡される引数の違いは、CPUが引数を関数にどのように引き渡すかを知らないという事実の良い例です。スタックをまったく使用せずに引数を特殊な構造体に渡すことができる仮想コンパイラを作成することもできます。

MIPS \$A0 ...\$A3 レジスタは、便宜上（O32呼び出し規約にある）このようにラベル付けされています。プログラマは、データを渡すために、あるいは他の呼び出し規約を使用するために、他のレジスタ（おそらく \$ZERO を除く）を使用することができます。

CPUは呼び出し規約を認識していません。

他の関数に引数を渡す新しいアセンブリ言語プログラマが、通常はレジスタ経由で、明示的な順序なしに、あるいはグローバル変数を介して、どのように新しい関数を呼び出すかを思い出すかもしれません。もちろん、それは正常に動作します。

第1.9節scanf()

では、scanf() を使ってみましょう。

第1.9.1節Simple example

```
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};
```

最近、ユーザーとのやり取りに scanf() を使用するの賢明ではありません。しかし、int 型の変数にポインタを渡す例で説明することができます。

ポインタについて

ポインタはコンピュータサイエンスの基本概念の1つです。多くの場合、大規模な配列、構造体またはオブジェクトを引数として別の関数に渡すことはコストがかかりすぎ、アドレスを渡すことはずっと安いです。たとえば、コンソールにテキスト文字列を印刷する場合、そのアドレスをOSカーネルに渡す方がはるかに簡単です。

さらに、[callee](#)関数が大きな配列または構造体の中の何かをパラメータとして受け取って構造体全体を返す必要がある場合、状況は不条理に近いです。したがって、配列や構造体のアドレスを呼び出し先関数に渡し、変更する必要があるものを変更させるのが最も簡単です。

C/C++ のポインタは、単純にあるメモリ位置のアドレスです。

x86では、アドレスは32ビット数（すなわち、4バイトを占める）で表され、x86-64では64ビット数（8バイトを占める）です。ところで、それがx86-64への切り替えに関連する憤慨の裏にある理由は、x64アーキテクチャのポインタは、「高価な」メモリであるキャッシュメモリを含めて、2倍のスペースを必要とします。

何らかの努力があれば、型の指定されていないポインタでのみ作業することができます。例えば1つのメモリ位置から別のメモリ位置にブロックをコピーする標準のC関数 memcpy() は、コピーするデータの型を予測することが不可能なため、void* 型の2つのポインタを引数としてとります。データ型は重要ではなく、ブロックサイズだけが重要です。

ポインタは、関数が複数の値を返す必要がある場合にも広く使用されます。(これについては後で説明します: [\(1.12 on page 138\)](#))

scanf() 関数が、このような場合です。

関数が正常に読み取られた値の数を示す必要があるという事実に加えて、これらの値もすべて返す必要があります。

C/C++ では、ポインタ型はコンパイル時の型チェックにのみ必要です。

内部的には、コンパイルされたコードには、ポインタ型に関する情報はまったくありません。

x86

MSVC

MSVC 2010でコンパイルした後に得られるものは次のとおりです。

```

CONST      SEGMENT
$SG3831     DB      'Enter X:', 0aH, 00H
$SG3832     DB      '%d', 00H
$SG3833     DB      'You entered %d...', 0aH, 00H
CONST      ENDS
PUBLIC      _main
EXTRN       _scanf:PROC
EXTRN       _printf:PROC
; 関数のコンパイルフラグ: /OdtP
_TEXT      SEGMENT
_x$ = -4                                ; size = 4
_main      PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    OFFSET $SG3831 ; 'Enter X:'
    call    _printf
    add     esp, 4
    lea     eax, DWORD PTR _x$[ebp]
    push    eax
    push    OFFSET $SG3832 ; '%d'
    call    _scanf
    add     esp, 8
    mov     ecx, DWORD PTR _x$[ebp]
    push    ecx
    push    OFFSET $SG3833 ; 'You entered %d...'
    call    _printf
    add     esp, 8

    ; 0をリターン
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main      ENDP

```

```
_TEXT    ENDS
```

x はローカル変数です。

C/C++ 標準によれば、この関数でのみ表示でき、他の外部スコープでは表示できません。従来、ローカル変数はスタックに格納されていました。それらを割り当てる方法はおそらく他にもありますが、それはx86の方法です。

関数プロログ、PUSH ECX に続く命令の目的は、ECX 状態を保存することではありません（関数の最後に対応する POP ECX が存在しないことに注意してください）。

実際、x 変数を格納するためにスタックに4バイトを割り当てます。

x は、_x\$ マクロ (-4に等しい) と現在のフレームを指す EBP レジスタの助けを借りてアクセスされます。

関数の実行の範囲にわたって、EBP は現在のstack frameを指しており、EBP+オフセットを介してローカル変数と関数引数にアクセスすることができます。

同じ目的で ESP を使用することもできますが、ESP は頻繁に変更されるためあまり便利ではありません。EBP の値は、関数の実行開始時に ESP の値が固定された状態として認識される可能性があります。

32ビット環境での典型的なstack frameレイアウトを次に示します。

...	...
EBP-8	local variable #2, IDA にマークする var_8
EBP-4	local variable #1, IDA にマークする var_4
EBP	saved value of EBP
EBP+4	return address
EBP+8	引数#1, IDA にマークする arg_0
EBP+0xC	引数#2, IDA にマークする arg_4
EBP+0x10	引数#3, IDA にマークする arg_8
...	...

この例の scanf() 関数には2つの引数があります。

最初のものは%d を含む文字列へのポインタで、2番目のものは x 変数のアドレスです。

最初に、x 変数のアドレスが lea eax, DWORD PTR _x\$[ebp] 命令によって EAX レジスタにロードされます。

LEA はロード実効アドレスの略で、アドレスを形成するためによく使用されます（(?? on page ??)）。

この場合、LEA は単に EBP レジスタ値と _x\$ マクロの合計を EAX レジスタに格納すると言うことができます。

これは lea eax, [ebp-4] と同じです。

したがって、EBP レジスタ値から4が減算され、その結果が EAX レジスタにロードされます。次に、EAX レジスタの値がスタックにプッシュされ、scanf() が呼び出されます。

printf() は最初の引数で呼び出されています。文字列へのポインタ：You entered %d...\n

2番目の引数は `mov ecx, [ebp-4]` で準備されています。命令は、ECX レジスタにそのアドレスではなく `x` 変数値を格納します。

次に、ECX の値がスタックに格納され、最後の `printf()` が呼び出されます。

MSVC + OllyDbg

OllyDbg でこの例を試してみましょう。私たちが `ntdll.dll` の代わりに実行可能ファイルに達するまで、それをロードしてF8（ステップオーバー）を押し続けましょう。`main()` が表示されるまで上にスクロールします。

最初の命令（`PUSH EBP`）をクリックし、F2（ブレークポイントを設定）、次にF9（実行）を押します。`main()` が始めるとブレークポイントがトリガされます。

変数 x のアドレスが計算されるポイントまでトレースしましょう：

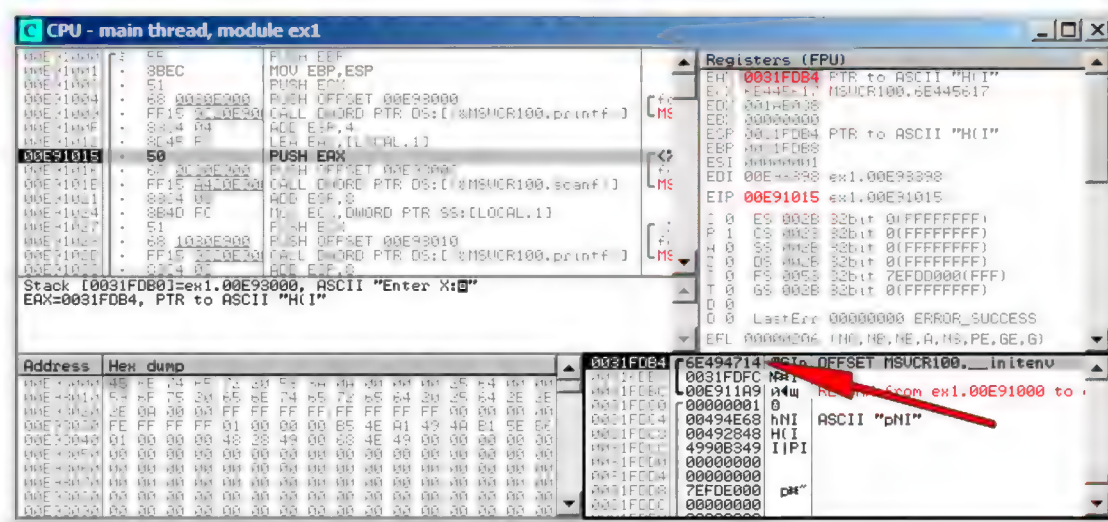


図 1.13: OllyDbg: ローカル変数のアドレスが計算されます。

レジスタウィンドウで EAX を右クリックして、「Follow in stack」を選択します。

このアドレスはスタックウィンドウに表示されます。赤い矢印が追加され、ローカルスタックの変数を指しています。その瞬間、この場所にはいくらかのゴミ（`0x6E494714`）が含まれています。今度は `PUSH` 命令の助けを借りて、このスタック要素のアドレスが次の位置の同じスタックに格納されます。`scanf()` の実行が完了するまで、F8を使ってトレースしてみましょう。`scanf()` の実行中に、コンソールウィンドウに123などを入力します。

Enter X:
123

scanf() はすでに実行を完了しました：

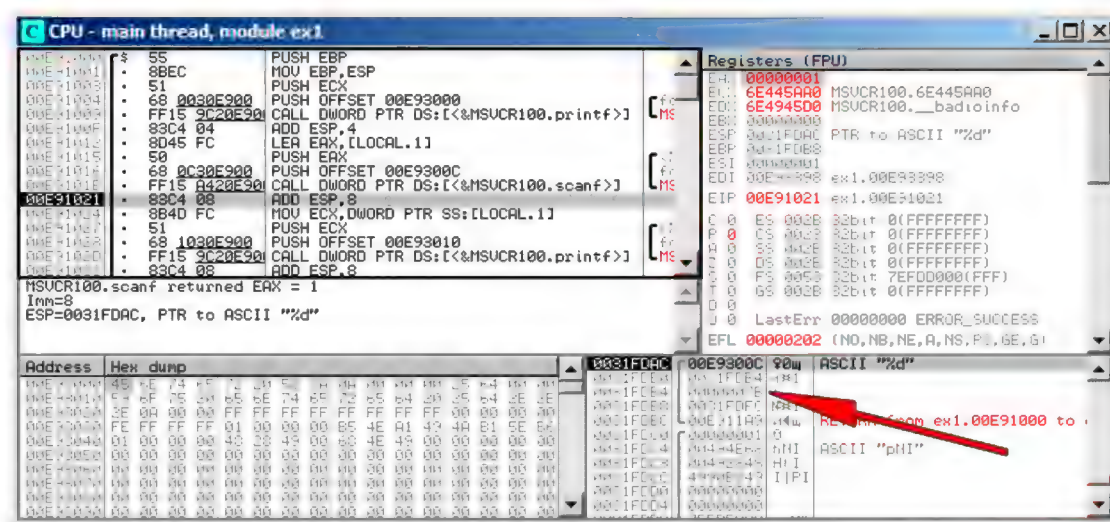


図 1.14: OllyDbg: scanf() が実行された

scanf() は EAX で1を返します。これは、1つの値を正常に読み取ったことを意味します。ローカル変数に対応するスタック要素をもう一度見ると、0x7B (123) が含まれています。

その後、この値はスタックから ECX レジスタにコピーされ、printf() に渡されます：

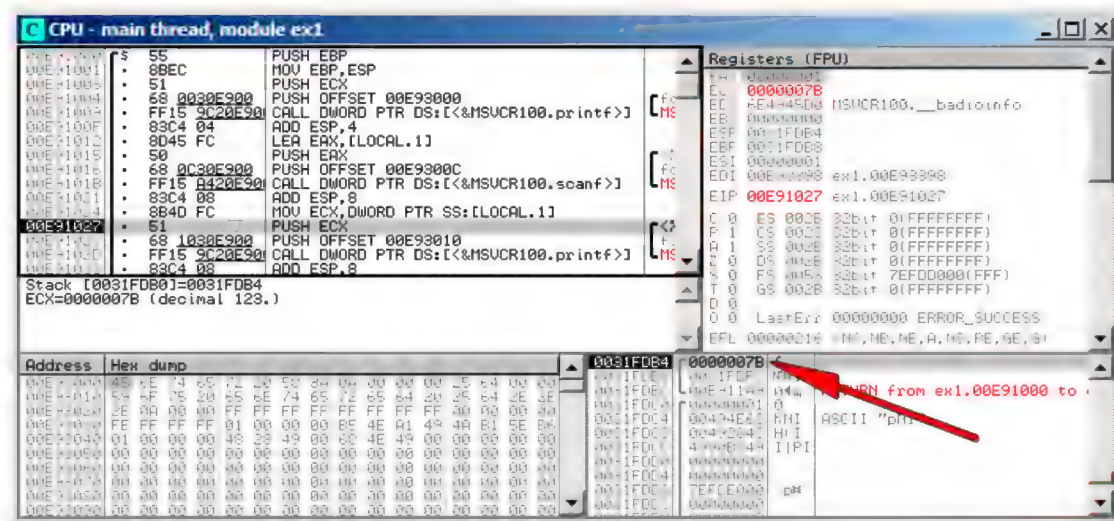


図 1.15: OllyDbg: printf() に渡す値を準備する

GCC

Linux上のGCC 4.4.1でこのコードをコンパイルしようとしましょう。

```

main      proc near
var_20    = dword ptr -20h
var_1C    = dword ptr -1Ch
var_4     = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 20h
        mov     [esp+20h+var_20], offset aEnterX ; "Enter X:"
        call    _puts
        mov     eax, offset aD ; "%d"
        lea     edx, [esp+20h+var_4]
        mov     [esp+20h+var_1C], edx
        mov     [esp+20h+var_20], eax
        call    __isoc99_scanf
        mov     edx, [esp+20h+var_4]
        mov     eax, offset aYouEnteredD ; "You entered %d...\n"
        mov     [esp+20h+var_1C], edx
        mov     [esp+20h+var_20], eax
        call    _printf
        mov     eax, 0
        leave

```

main	retn endp
------	--------------

GCCは `printf()` 呼び出しを `puts()` の呼び出しで置き換えました。この理由は、[\(1.5.3 on page 27\)](#) で説明されました。

MSVCの例のように、引数は `MOV` 命令を使用してスタックに配置されます。

ところで

ところで、この単純な例は、コンパイラが C/C++ ブロックの式のリストを命令の連続したリストに変換するという事実のデモンストレーションです。C/C++ の式の間には何もないので、結果のマシンコードには、ある式から次の式への制御フローの間には何もありません。

x64

この画像は、スタックではなくレジスタが引数の受け渡しに使用されるという違いと似ています。

MSVC

Listing 1.67: MSVC 2012 x64

```

_DATA    SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA    ENDS

_TEXT    SEGMENT
x$ = 32
main     PROC
$LN3:
    sub     rsp, 56
    lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
    call    printf
    lea     rdx, QWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
    call    scanf
    mov     edx, DWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
    call    printf

    ; return 0
    xor     eax, eax
    add     rsp, 56
    ret     0
main     ENDP
_TEXT    ENDS

```

GCC

Listing 1.68: 最適化 GCC 4.4.6 x64

```

.LC0:
    .string "Enter X:"
.LC1:
    .string "%d"
.LC2:
    .string "You entered %d...\n"
main:
    sub     rsp, 24
    mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
    call    puts
    lea     rsi, [rsp+12]
    mov     edi, OFFSET FLAT:.LC1 ; "%d"
    xor     eax, eax
    call    __isoc99_scanf
    mov     esi, DWORD PTR [rsp+12]
    mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
    xor     eax, eax
    call    printf

    ; return 0
    xor     eax, eax
    add     rsp, 24
    ret

```

ARM

最適化 Keil 6/2013 (Thumbモード)

```

.text:00000042      scanf_main
.text:00000042
.text:00000042      var_8          = -8
.text:00000042
.text:00000042 08 B5      PUSH    {R3,LR}
.text:00000044 A9 A0      ADR     R0, aEnterX ; "Enter X:\n"
.text:00000046 06 F0 D3 F8  BL      __2printf
.text:0000004A 69 46      MOV     R1, SP
.text:0000004C AA A0      ADR     R0, aD ; "%d"
.text:0000004E 06 F0 CD F8  BL      __0scanf
.text:00000052 00 99      LDR     R1, [SP,#8+var_8]
.text:00000054 A9 A0      ADR     R0, aYouEnteredD____ ; "You entered
    %d...\n"
.text:00000056 06 F0 CB F8  BL      __2printf
.text:0000005A 00 20      MOVS    R0, #0
.text:0000005C 08 BD      POP     {R3,PC}

```

scanf() がitemを読み込むためには、*int* へのparameter.pointerが必要です。*int* は32ビットなので、メモリのどこかに格納するには4バイトが必要で、32ビットのレジスタに正

確に収まります。ローカル変数 `x` の場所がスタックに割り当てられ、IDA の名前は `var_8` です。ただし、**SP!** ([スタックポインタ](#)) がすでにその領域を指しているため、その領域を直接割り当てることはできません。

PUSH/POP 命令は、ARMとx86とでは動作が異なります（これは逆です）。これらは STM/STMDB/LDM/LDMIA 命令の同義語です。そして、PUSH 命令は最初に値をスタックに書き込み、次に **SP!** を4で減算します。POP は最初に **SP!** に4を加算してから、スタックから値を読み取ります。したがって、PUSH 後、**SP!** はスタック内の未使用スペースを指します。それは `scanf()` によって、そして後に `printf()` によって使用されます。

LDMIA は *Load Multiple Registers Increment address After each transfer* の略です。STMDB は *Store Multiple Registers Decrement address Before each transfer* の略です。

したがって、**SP!** の値は R1 レジスタにコピーされ、フォーマット文字列とともに `scanf()` に渡されます。その後、LDR 命令の助けを借りて、この値はスタックから R1 レジスタに移動され、`printf()` に渡されます。

ARM64

Listing 1.69: 非最適化 GCC 4.9.1 ARM64

```

1  .LC0:
2      .string "Enter X:"
3  .LC1:
4      .string "%d"
5  .LC2:
6      .string "You entered %d...\n"
7  scanf_main:
8      ; subtract 32 from SP, then save FP and LR in stack frame:
9      stp    x29, x30, [sp, -32]!
10     ; set stack frame (FP=SP)
11     add    x29, sp, 0
12     ; load pointer to the "Enter X:" string:
13     adrp   x0, .LC0
14     add    x0, x0, :lo12:LC0
15     ; X0=pointer to the "Enter X:" string
16     ; print it:
17     bl     puts
18     ; load pointer to the "%d" string:
19     adrp   x0, .LC1
20     add    x0, x0, :lo12:LC1
21     ; find a space in stack frame for "x" variable (X1=FP+28):
22     add    x1, x29, 28
23     ; X1=address of "x" variable
24     ; pass the address to scanf() and call it:
25     bl     __isoc99_scanf
26     ; load 32-bit value from the variable in stack frame:
27     ldr    w1, [x29,28]
28     ; W1=x
29     ; load pointer to the "You entered %d...\n" string
30     ; printf() will take text string from X0 and "x" variable from X1 (or W1)
31     adrp   x0, .LC2
32     add    x0, x0, :lo12:LC2

```

```

33      bl      printf
34 ; return 0
35      mov     w0, 0
36 ; restore FP and LR, then add 32 to SP:
37      ldp     x29, x30, [sp], 32
38      ret

```

スタックフレームには32バイトが割り当てられており、必要なサイズよりも大きくなっています。たぶんメモリのアラインメントの問題でしょうか？最も興味深いのはスタックフレーム内の x 変数のためのスペースを見つけることです（22行目）。なぜ28なのでしょう？何らかの理由で、コンパイラは、この変数をスタックフレームの最後に置きます。アドレスは `scanf()` に渡され、`scanf()` はユーザ入力値をそのアドレスのメモリに格納するだけです。これは `int` 型の32ビット値です。値は27行目から取得され、`printf()` に渡されます。

MIPS

ローカルスタック内の場所は x 変数に割り当てられ、 $\$sp + 24$ と呼ばれます。

そのアドレスは `scanf()` に渡され、ユーザー入力値は `INCLW`（「Load Word」）を使用してロードされます。そしてそれから `printf()` に渡されます。

Listing 1.70: 最適化 GCC 4.4.5 (アセンブリ出力)

```

$LC0:
    .ascii  "Enter X:\000"
$LC1:
    .ascii  "%d\000"
$LC2:
    .ascii  "You entered %d...\012\000"
main:
; 関数プロローグ:
    lui     $28,%hi(__gnu_local_gp)
    addiu   $sp,$sp,-40
    addiu   $28,$28,%lo(__gnu_local_gp)
    sw      $31,36($sp)
; puts() を呼び出す:
    lw      $25,%call16(puts)($28)
    lui     $4,%hi($LC0)
    jalr    $25
    addiu   $4,$4,%lo($LC0) ; branch delay slot
; scanf() を呼び出す:
    lw      $28,16($sp)
    lui     $4,%hi($LC1)
    lw      $25,%call16(__isoc99_scanf)($28)
; scanf() の2番目の引数に $a1=$sp+24 を設定する:
    addiu   $5,$sp,24
    jalr    $25
    addiu   $4,$4,%lo($LC1) ; branch delay slot

; printf() を呼び出す:
    lw      $28,16($sp)
; printf() の2番目の引数を設定する,

```

```

; アドレス $sp+24にwordをロードする:
    lw      $5,24($sp)
    lw      $25,%call16(sprintf)($28)
    lui     $4,%hi($LC2)
    jalr    $25
    addiu   $4,$4,%lo($LC2) ; branch delay slot

; 関数エピローグ:
    lw      $31,36($sp)
; 戻り値に0を設定する:
    move    $2,$0
; return:
    j       $31
    addiu   $sp,$sp,40      ; branch delay slot

```

IDAはスタックレイアウトを次のように表示します。

Listing 1.71: 最適化 GCC 4.4.5 (IDA)

```

.text:00000000 main:
.text:00000000
.text:00000000 var_18    = -0x18
.text:00000000 var_10    = -0x10
.text:00000000 var_4     = -4
.text:00000000
; 関数プロローグ:
.text:00000000          lui     $gp, (__gnu_local_gp >> 16)
.text:00000004          addiu   $sp, -0x28
.text:00000008          la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000000C          sw      $ra, 0x28+var_4($sp)
.text:00000010          sw      $gp, 0x28+var_18($sp)
; puts() を呼び出す:
.text:00000014          lw      $t9, (puts & 0xFFFF)($gp)
.text:00000018          lui     $a0, ($LC0 >> 16) # "Enter X:"
.text:0000001C          jalr    $t9
.text:00000020          la      $a0, ($LC0 & 0xFFFF) # "Enter X:" ; branch
                    delay slot
; scanf() を呼び出す:
.text:00000024          lw      $gp, 0x28+var_18($sp)
.text:00000028          lui     $a0, ($LC1 >> 16) # "%d"
.text:0000002C          lw      $t9, (__isoc99_scanf & 0xFFFF)($gp)
; scanf() の2番目の引数に $a1=$sp+24 を設定する:
.text:00000030          addiu   $a1, $sp, 0x28+var_10
.text:00000034          jalr    $t9 ; branch delay slot
.text:00000038          la      $a0, ($LC1 & 0xFFFF) # "%d"
; printf() を呼び出す:
.text:0000003C          lw      $gp, 0x28+var_18($sp)
; printf() の2番目の引数を設定する,
; アドレス $sp+24にwordをロードする:
.text:00000040          lw      $a1, 0x28+var_10($sp)
.text:00000044          lw      $t9, (printf & 0xFFFF)($gp)
.text:00000048          lui     $a0, ($LC2 >> 16) # "You entered %d...\n"
.text:0000004C          jalr    $t9
.text:00000050          la      $a0, ($LC2 & 0xFFFF) # "You entered %d...\n"

```

```

; branch delay slot
; 関数エピローグ:
.text:00000054      lw      $ra, 0x28+var_4($sp)
; 戻り値に0を設定する:
.text:00000058      move    $v0, $zero
; return:
.text:0000005C      jr      $ra
.text:00000060      addiu   $sp, 0x28 ; branch delay slot

```

第1.9.2節一般的な間違い

x へのポインタではなく、 x の値を渡すのは極めて一般的な間違い（および/またはタイプミス）です。

```

#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    scanf ("%d", x); // BUG

    printf ("You entered %d...\n", x);

    return 0;
};

```

では、何が起こるのでしょうか？ x は初期化されておらず、ローカルスタックからのランダムノイズを含んでいます。scanf() が呼び出されると、ユーザーから文字列を受け取り、数値に解析し、 x に書き込んでメモリ内のアドレスとして扱います。しかしランダムなノイズがあるので、scanf() はランダムなアドレスに書き込もうとします。おそらく、プロセスがクラッシュするでしょう。

興味深いことに、デバッグビルドのいくつかのCRTライブラリは、視覚的に特徴的なパターンを0xCCCCCCCCや0x0BADF00Dのように割り当てられたメモリに入れています。この場合、 x は0xCCCCCCCCを含むことができ、scanf() はアドレス0xCCCCCCCCに書き込みを試みます。また、プロセス内の何かがアドレス0xCCCCCCCCに書き込もうとすると、初期化されていない変数（またはポインタ）が事前初期化なしで使用されることがわかります。これは、新しく割り当てられたメモリがちょうどクリアされた場合よりも優れています。

第1.9.3節グローバル変数

前の例の x 変数がローカルではなく、グローバル変数であればどうでしょうか？それから、関数本体からだけでなく、どの時点からでもアクセスできるようになりました。グローバル変数は[anti-pattern](#)と見なされますが、実験のために行ってみましょう。

```

#include <stdio.h>

// now x is global variable

```



```

int x;

int main()
{
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};

```

MSVC: x86

```

_DATA    SEGMENT
COMM     _x:DWORD
$SG2456  DB      'Enter X:', 0aH, 00H
$SG2457  DB      '%d', 00H
$SG2458  DB      'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC   _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
    push  ebp
    mov   ebp, esp
    push  OFFSET $SG2456
    call  _printf
    add   esp, 4
    push  OFFSET _x
    push  OFFSET $SG2457
    call  _scanf
    add   esp, 8
    mov   eax, DWORD PTR _x
    push  eax
    push  OFFSET $SG2458
    call  _printf
    add   esp, 8
    xor   eax, eax
    pop   ebp
    ret   0
_main    ENDP
_TEXT    ENDS

```

この場合、`x` 変数は `_DATA` セグメントに定義され、ローカルスタックにはメモリは割り当てられません。スタックからではなく、直接アクセスされます。初期化されていないグローバル変数は、実行可能ファイルにスペースを入れません（なぜ、最初に変数をゼロに設定する必要があるのでしょうか?）。しかし、誰かが自分のアドレスにアクセスすると、

OSは0で初期化されたブロック⁷⁰を割り当てます。

変数に明示的に値を割り当てましょう：

```
int x=10; // default value
```

以下を得ます。

_DATA	SEGMENT	
_x	DD	0aH
...		

ここでは、この変数のDWORDタイプの値 0xA (DDはDWORD = 32ビットを表します)が表示されます。

IDA にコンパイルされた.exeを開くと、_DATA セグメントの先頭に x 変数が配置されていて、その後にテキスト文字列が表示されます。

x の値が設定されていない前の例のコンパイル済み.exeを IDA で開くと、次のように表示されます。

Listing 1.72: IDA

.data:0040FA80 _x	dd ?	; DATA XREF: _main+10
.data:0040FA80		; _main+22
.data:0040FA84 dword_40FA84	dd ?	; DATA XREF: _memset+1E
.data:0040FA84		; unknown_libname_1+28
.data:0040FA88 dword_40FA88	dd ?	; DATA XREF: ___sbh_find_block+5
.data:0040FA88		; ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem		
.data:0040FA8C lpMem	dd ?	; DATA XREF: ___sbh_find_block+B
.data:0040FA8C		; ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90	dd ?	; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90		; __calloc_impl+72
.data:0040FA94 dword_40FA94	dd ?	; DATA XREF: ___sbh_free_block+2FE

_x に? がマークされていると、残りの変数は初期化する必要はありません。これは、メモリに.exeをロードした後、これらすべての変数のための領域が割り当てられ、0で満たされる [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.8p10] ことを意味します。しかし、.exeファイルでは、これらの初期化されていない変数は何も占有しません。これは、例えば、大きな配列の場合に便利です。

⁷⁰これがVM⁷¹の動作です

MSVC: x86 + OllyDbg

ここではさらに単純です。

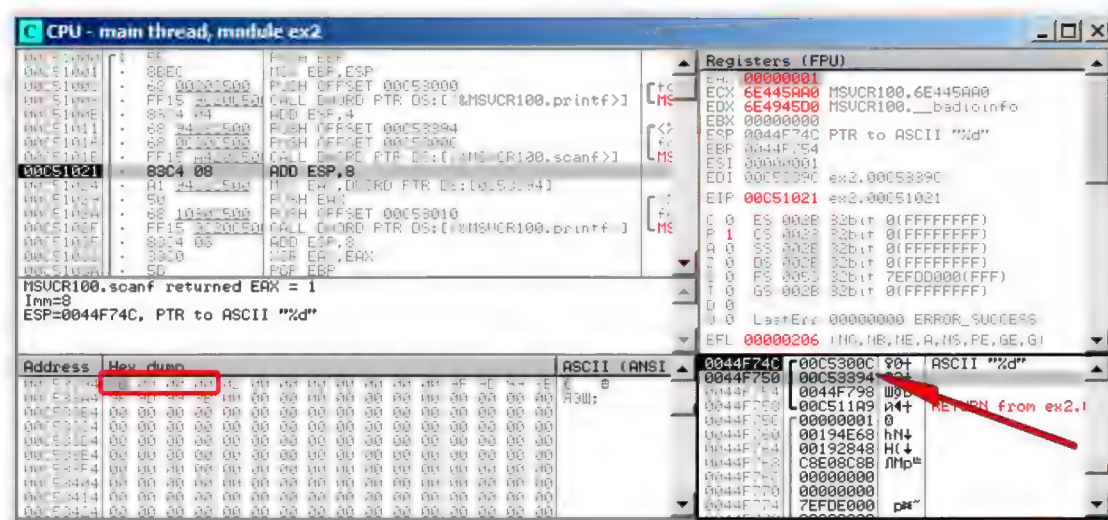


図 1.16: OllyDbg: after scanf() execution

変数はデータセグメントにあります。PUSH 命令 (x のアドレスを押す) が実行されると、アドレスがスタックウィンドウに表示されます。その行を右クリックし、「ダンプに従う」を選択します。変数は、左側のメモリウィンドウに表示されます。コンソールに123を入力すると、メモリウィンドウに 0x7B が表示されます (ハイライトされたスクリーンショット領域を参照)。

しかし、最初のバイトはなぜ7Bでしょうか？論理的に考えると、00 00 00 7B のはずで、この原因は **endianness** と呼ばれるもので、x86 はリトルエンディアンを使用します。これは、最下位バイトが最初に書き込まれ、最上位バイトが最後に書き込まれることを意味します。これについての詳細: ?? on page ?? この例では、32ビットの値がこのメモリアドレスから EAX にロードされ、printf() に渡されます。

x のメモリアドレスは 0x00C53394 です。

OllyDbg では、プロセスメモリマップ (Alt-M) を見ることができ、このアドレスはプログラムの .data PEセグメント内にあることがわかります。

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00070000	00057000			Heap	Map	R	R	C:\Windows\System32\locale.nls
00190000	00005000				Priv	RW	RW	
00209000	00007000				Priv	RW	Gua: RW	Gua:
0044C000	00001000				Priv	RW	Gua: RW	Gua:
0044D000	00003000			Stack of main thread	Priv	RW	RW	
00590000	00007000				Priv	RW	RW	
00750000	0000C000			Default heap	Priv	RW	RW	
00C50000	00001000	ex2		PE header	Img	R	RWE	Cop:
00C51000	00001000	ex2	.text	Code	Img	R E	RWE	Cop:
00C52000	00001000	ex2	.rdata	Imports	Img	R	RWE	Cop:
00C53000	00001000	ex2	.data	Data	Img	RW	RWE	Cop:
00C54000	00001000	ex2	.reloc	Relocations	Img	R	RWE	Cop:
6E3E0000	00001000	MSUCR100		PE header	Img	R	RWE	Cop:
6E3E1000	00002000	MSUCR100	.text	Code, imports, exports	Img	R E	RWE	Cop:
6E493000	00006000	MSUCR100	.data	Data	Img	RW	Cop: RWE	Cop:
6E499000	00001000	MSUCR100	.rsrc	Resources	Img	R	RWE	Cop:
6E49A000	00005000	MSUCR100	.reloc	Relocations	Img	R	RWE	Cop:
755D0000	00001000	Mod_755D		PE header	Img	R	RWE	Cop:
755D1000	00003000				Img	R E	RWE	Cop:
755D4000	00001000				Img	RW	RWE	Cop:
755D5000	00003000				Img	R	RWE	Cop:
755E0000	00001000	Mod_755E		PE header	Img	R	RWE	Cop:
755E1000	00004000				Img	R E	RWE	Cop:
7562E000	00005000				Img	RW	Cop: RWE	Cop:
75633000	00009000				Img	R	RWE	Cop:
75640000	00001000	Mod_7564		PE header	Img	R	RWE	Cop:
75641000	00003000				Img	R E	RWE	Cop:
75679000	00002000				Img	RW	RWE	Cop:
7567B000	00004000				Img	R	RWE	Cop:
76F50000	00001000	kernel32		PE header	Img	R	RWE	Cop:
76F60000	0000D000	kernel32	.text	Code, imports, exports	Img	R E	RWE	Cop:
77030000	00001000	kernel32	.data	Data	Img	RW	Cop: RWE	Cop:
77040000	00001000	kernel32	.rsrc	Resources	Img	R	RWE	Cop:
77050000	0000B000	kernel32	.reloc	Relocations	Img	R	RWE	Cop:
77810000	00001000	KERNELBASE		PE header	Img	R	RWE	Cop:
77811000	00004000	KERNELBASE	.text	Code, imports, exports	Img	R E	RWE	Cop:
77851000	00002000	KERNELBASE	.data	Data	Img	RW	RWE	Cop:
77853000	00001000	KERNELBASE	.rsrc	Resources	Img	R	RWE	Cop:
77854000	00003000	KERNELBASE	.reloc	Relocations	Img	R	RWE	Cop:
77B20000	00001000	Mod_77B2		PE header	Img	R	RWE	Cop:
77B21000	00102000				Img	R E	RWE	Cop:
77C23000	0002F000				Img	R	RWE	Cop:
77C52000	0000C000				Img	RW	Cop: RWE	Cop:
77C5E000	0000B000				Img	R	RWE	Cop:
77D00000	00001000	ntdll		PE header	Img	R	RWE	Cop:
77D10000	00005000	ntdll	.text	Code, exports	Img	R E	RWE	Cop:
77DF0000	00001000	ntdll	.rsrc	Resources	Img	R	RWE	Cop:
77E00000	00003000	ntdll	.data	Data	Img	RW	Cop: RWE	Cop:

図 1.17: OllyDbg: process memory map

GCC: x86

Linuxの画像はほぼ同じですが、初期化されていない変数は `_bss` セグメントにあります。[ELF⁷²](#) ファイルでは、このセグメントには次の属性があります。

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

ただし、変数のある値で初期化してください。10の場合、次の属性を持つ `_data` セグメントに配置されます。

```
; Segment type: Pure data
; Segment permissions: Read/Write
```

MSVC: x64

⁷² Executable and Linkable Format: Linuxを含め*NIXシステムで広く使用される実行ファイルフォーマット

Listing 1.73: MSVC 2012 x64

```

_DATA    SEGMENT
_COMM    x:DWORD
$SG2924  DB      'Enter X:', 0aH, 00H
$SG2925  DB      '%d', 00H
$SG2926  DB      'You entered %d...', 0aH, 00H
_DATA    ENDS

_TEXT    SEGMENT
main     PROC
$LN3:

    sub     rsp, 40

    lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call    printf
    lea     rdx, OFFSET FLAT:x
    lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
    call    scanf
    mov     edx, DWORD PTR x
    lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
    call    printf

    ; 0をリターンする
    xor     eax, eax

    add     rsp, 40
    ret     0
main     ENDP
_TEXT    ENDS

```

コードはx86とほとんど同じです。 x 変数のアドレスは、LEA 命令を使用して `scanf()` に渡され、変数の値は MOV 命令を使用して2番目の `printf()` に渡されることに注意してください。DWORD PTR はアセンブリ言語の一部であり（マシンコードと無関係）、可変データサイズが32ビットであり、MOV 命令がそれに応じてエンコードされなければならないことを示します。

ARM: 最適化 Keil 6/2013 (Thumbモード)

Listing 1.74: IDA

```

.text:00000000 ; Segment type: Pure code
.text:00000000     AREA .text, CODE
...
.text:00000000 main
.text:00000000     PUSH     {R4,LR}
.text:00000002     ADR      R0, aEnterX ; "Enter X:\n"
.text:00000004     BL       __2printf
.text:00000008     LDR      R1, =x
.text:0000000A     ADR      R0, aD ; "%d"
.text:0000000C     BL       __0scanf
.text:00000010     LDR      R0, =x
.text:00000012     LDR      R1, [R0]

```

```

.text:00000014      ADR      R0, aYouEnteredD____ ; "You entered %d...\n"
.text:00000016      BL       __2printf
.text:0000001A      MOVS     R0, #0
.text:0000001C      POP      {R4,PC}
...
.text:00000020 aEnterX DCB "Enter X:",0xA,0 ; DATA XREF: main+2
.text:0000002A      DCB      0
.text:0000002B      DCB      0
.text:0000002C off_2C DCD x ; DATA XREF: main+8
.text:0000002C ; main+10
.text:00000030 aD      DCB "%d",0 ; DATA XREF: main+A
.text:00000033      DCB      0
.text:00000034 aYouEnteredD____ DCB "You entered %d...",0xA,0 ; DATA XREF:
main+14
.text:00000047      DCB      0
.text:00000047 ; .text ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048      AREA .data, DATA
.data:00000048 ; ORG 0x48
.data:00000048      EXPORT x
.data:00000048 x      DCD 0xA ; DATA XREF: main+8
.data:00000048 ; main+10
.data:00000048 ; .data ends

```

したがって、x 変数は現在グローバルであり、このために別のセグメント、つまりデータセグメント (.data) に配置されています。テキスト文字列がコードセグメント (.text) にあり、x がここにあるのはなぜでしょうか？これは変数なので、定義上、その値は変更される可能性があります。さらに、頻繁に変更される可能性があります。テキスト文字列は定数型ですが、変更されないため、.text セグメントに配置されます。

コードセグメントは、時にはROM⁷³チップに配置されることがあります。(ここでは、組み込み電子機器を扱います。メモリ不足が普通です) 変更可能な変数はRAMに配置されます。

ROMを持っているときは、定数変数をRAMに格納するのはそれほど経済的ではありません。

さらに、RAMの定数変数は初期化する必要があります。これは、電源投入後、明らかにRAMにランダム情報が含まれているためです。

次に、コードセグメント内の x (off_2C) 変数へのポインターが表示され、変数を使用するすべての操作はこのポインターを介して行われます。

これは、x 変数がこの特定のコードフラグメントから離れた場所に配置される可能性があるため、そのアドレスをコードのすぐ近くに保存する必要があるためです。

Thumbモードの LDR 命令は、その位置から1020バイトの範囲内の変数と、±4095 バイトの範囲のARMモードの変数からのみアドレス可能です。

したがって、x 変数のアドレスは、リンカがコードの近くのどこかに変数を格納できる保証がないため、近い場所に配置する必要があります。外部メモリチップでもうまくいかかもしれません！

⁷³読み取り専用メモリ

もう1つ：変数が *const* として宣言されている場合、Keilコンパイラは *.constdata* セグメントにそれを割り当てます。

その後、リンカはこのセグメントをROMにコードセグメントとともに配置することができます。

ARM64

Listing 1.75: 非最適化 GCC 4.9.1 ARM64

```

1      .comm    x,4,4
2      .LC0:
3          .string "Enter X:"
4      .LC1:
5          .string "%d"
6      .LC2:
7          .string "You entered %d...\n"
8      f5:
9      ; FPとLRをスタックフレームに保存する:
10         stp    x29, x30, [sp, -16]!
11      ; スタックフレームを設定する (FP=SP)
12         add    x29, sp, 0
13      ; "Enter X:" 文字列へのポインタをロードする:
14         adrp   x0, .LC0
15         add    x0, x0, :lo12:LC0
16         bl     puts
17      ; "%d" 文字列へのポインタをロードする:
18         adrp   x0, .LC1
19         add    x0, x0, :lo12:LC1
20      ; xグローバル変数のアドレスを形作る:
21         adrp   x1, x
22         add    x1, x1, :lo12:x
23         bl     __isoc99_scanf
24      ; 再度xグローバル変数のアドレスを形作る:
25         adrp   x0, x
26         add    x0, x0, :lo12:x
27      ; このアドレスのメモリから値をロードする:
28         ldr     w1, [x0]
29      ; "You entered %d...\n" 文字列へのポインタをロードする:
30         adrp   x0, .LC2
31         add    x0, x0, :lo12:LC2
32         bl     printf
33      ; 0をリターンする
34         mov     w0, 0
35      ; FPとLRをリストアする:
36         ldp     x29, x30, [sp], 16
37         ret

```

この場合、*x* 変数はグローバルとして宣言され、そのアドレスは *ADRP/ADD* 命令ペア (21行目と25行目) を使用して計算されます。

MIPS

初期化されていないグローバル変数

だから今 x 変数はグローバルです。オブジェクトファイルではなく実行ファイルにコンパイルし、IDA にロードしてみましょう。IDAは、`.sbss` ELFセクションに x 変数を表示します（25ページの「グローバルポインタ」 [1.5.4 on page 32](#)を覚えておいてください）。これは変数が最初に初期化されていないためです。

Listing 1.76: 最適化 GCC 4.4.5 (IDA)

```
.text:004006C0 main:
.text:004006C0
.text:004006C0 var_10 = -0x10
.text:004006C0 var_4 = -4
.text:004006C0
; 関数プロローグ:
.text:004006C0      lui      $gp, 0x42
.text:004006C4      addiu    $sp, -0x20
.text:004006C8      li       $gp, 0x418940
.text:004006CC      sw       $ra, 0x20+var_4($sp)
.text:004006D0      sw       $gp, 0x20+var_10($sp)
; puts() を呼び出す:
.text:004006D4      la       $t9, puts
.text:004006D8      lui      $a0, 0x40
.text:004006DC      jalr     $t9 ; puts
.text:004006E0      la       $a0, aEnterX      # "Enter X:" ; branch delay
slot
; scanf() を呼び出す:
.text:004006E4      lw       $gp, 0x20+var_10($sp)
.text:004006E8      lui      $a0, 0x40
.text:004006EC      la       $t9, __isoc99_scanf
; xのアドレスを準備する:
.text:004006F0      la       $a1, x
.text:004006F4      jalr     $t9 ; __isoc99_scanf
.text:004006F8      la       $a0, aD      # "%d" ; branch delay slot
; printf() を呼び出す:
.text:004006FC      lw       $gp, 0x20+var_10($sp)
.text:00400700      lui      $a0, 0x40
; xのアドレスを取得する:
.text:00400704      la       $v0, x
.text:00400708      la       $t9, printf
; 変数xから値をロードして $a1にてprintf() に値を渡す:
.text:0040070C      lw       $a1, (x - 0x41099C)($v0)
.text:00400710      jalr     $t9 ; printf
.text:00400714      la       $a0, aYouEnteredD__ # "You entered %d...\n"
; branch delay slot
; 関数エピローグ:
.text:00400718      lw       $ra, 0x20+var_4($sp)
.text:0040071C      move    $v0, $zero
.text:00400720      jr      $ra
.text:00400724      addiu    $sp, 0x20 ; branch delay slot
...

```

```
.sbss:0041099C # Segment type: Uninitialized
.sbss:0041099C .sbss
.sbss:0041099C .globl x
.sbss:0041099C x: .space 4
.sbss:0041099C
```

IDAは情報量を減らすため、objdumpを使用してリスティングを行い、コメントします。

Listing 1.77: 最適化 GCC 4.4.5 (objdump)

```
1 004006c0 <main>:
2 ; 関数プロローグ:
3 4006c0: 3c1c0042 lui gp,0x42
4 4006c4: 27bdf0e0 addiu sp,sp,-32
5 4006c8: 279c8940 addiu gp,gp,-30400
6 4006cc: afbf001c sw ra,28(sp)
7 4006d0: afbc0010 sw gp,16(sp)
8 ; puts() を呼び出す:
9 4006d4: 8f998034 lw t9,-32716(gp)
10 4006d8: 3c040040 lui a0,0x40
11 4006dc: 0320f809 jalr t9
12 4006e0: 248408f0 addiu a0,a0,2288 ; branch delay slot
13 ; scanf() を呼び出す:
14 4006e4: 8fbc0010 lw gp,16(sp)
15 4006e8: 3c040040 lui a0,0x40
16 4006ec: 8f998038 lw t9,-32712(gp)
17 ; xのアドレスを準備する:
18 4006f0: 8f858044 lw a1,-32700(gp)
19 4006f4: 0320f809 jalr t9
20 4006f8: 248408fc addiu a0,a0,2300 ; branch delay slot
21 ; printf() を呼び出す:
22 4006fc: 8fbc0010 lw gp,16(sp)
23 400700: 3c040040 lui a0,0x40
24 ; xのアドレスを取得する:
25 400704: 8f828044 lw v0,-32700(gp)
26 400708: 8f99803c lw t9,-32708(gp)
27 ; 変数xから値をロードして $a1にてprintf() に値を渡す:
28 40070c: 8c450000 lw a1,0(v0)
29 400710: 0320f809 jalr t9
30 400714: 24840900 addiu a0,a0,2304 ; branch delay slot
31 ; 関数エピローグ:
32 400718: 8fbf001c lw ra,28(sp)
33 40071c: 00001021 move v0,zero
34 400720: 03e00008 jr ra
35 400724: 27bd0020 addiu sp,sp,32 ; branch delay slot
36 ; 次の関数の開始アドレスが16バイト境界になるようにNOPで埋める:
37 400728: 00200825 move at,at
38 40072c: 00200825 move at,at
```

今度は *x* 変数アドレスがGPを使って64KiBのデータバッファから読み込まれ、負のオフセットが加えられていることがわかります (18行目)。さらに、この例 (puts()、scanf()、printf()) で使用されている3つの外部関数のアドレスもGPを使用して64KiBグロー

バルデータバッファから読み込まれます (9,16,26行目)。GPはバッファの中央を指しています。このようなオフセットは、3つの関数のアドレスと x 変数のアドレスがすべてそのバッファの先頭に格納されていることを示しています。私たちの例は非常に小さいので、それは理にかなっています。

言及する価値がある別のことは、次の関数の開始を16バイトの境界に合わせるために、関数が2つのNOP (MOVE \$AT,\$AT、アイドル命令) で終了することです。

初期化されたグローバル変数

x 変数にデフォルト値を与えることで、この例を変更しましょう。

```
int x=10; // default value
```

IDAは x 変数が.dataセクションに存在することを示しています：

Listing 1.78: 最適化 GCC 4.4.5 (IDA)

```
.text:004006A0 main:
.text:004006A0
.text:004006A0 var_10 = -0x10
.text:004006A0 var_8 = -8
.text:004006A0 var_4 = -4
.text:004006A0
.text:004006A0      lui      $gp, 0x42
.text:004006A4      addiu    $sp, -0x20
.text:004006A8      li       $gp, 0x418930
.text:004006AC      sw       $ra, 0x20+var_4($sp)
.text:004006B0      sw       $s0, 0x20+var_8($sp)
.text:004006B4      sw       $gp, 0x20+var_10($sp)
.text:004006B8      la       $t9, puts
.text:004006BC      lui      $a0, 0x40
.text:004006C0      jalr     $t9 ; puts
.text:004006C4      la       $a0, aEnterX      # "Enter X:"
.text:004006C8      lw       $gp, 0x20+var_10($sp)
; アドレスxの高ビットを準備する:
.text:004006CC      lui      $s0, 0x41
.text:004006D0      la       $t9, __isoc99_scanf
.text:004006D4      lui      $a0, 0x40
; アドレスxの低ビットを準備する:
.text:004006D8      addiu    $a1, $s0, (x - 0x410000)
; アドレスxは $a1にあります
.text:004006DC      jalr     $t9 ; __isoc99_scanf
.text:004006E0      la       $a0, aD      # "%d"
.text:004006E4      lw       $gp, 0x20+var_10($sp)
; メモリからwordを取得する:
.text:004006E8      lw       $a1, x
; xの値は $a1にあります
.text:004006EC      la       $t9, printf
.text:004006F0      lui      $a0, 0x40
.text:004006F4      jalr     $t9 ; printf
.text:004006F8      la       $a0, aYouEnteredD      # "You entered %d...\n"
.text:004006FC      lw       $ra, 0x20+var_4($sp)
```

```
.text:00400700      move    $v0, $zero
.text:00400704      lw      $s0, 0x20+var_8($sp)
.text:00400708      jr      $ra
.text:0040070C      addiu   $sp, 0x20

...

.data:00410920      .globl x
.data:00410920 x:   .word 0xA
```

.sdataにしたら？これはおそらくいくつかのGCCオプションに依存するのでしょうか？

それにもかかわらず、 x は一般的なメモリ領域である.dataにあり、ここで変数を扱う方法を見てみるすることができます。

変数のアドレスは、命令のペアを使用して構成する必要があります。

私たちの場合、それらは LUI (「Load Upper Immediate」) と ADDIU (「Add Immediate Unsigned Word」) です。

厳密な検査のためのobjdumpリストもあります：

Listing 1.79: 最適化 GCC 4.4.5 (objdump)

```
004006a0 <main>:
 4006a0: 3c1c0042 lui      gp,0x42
 4006a4: 27bdffe0 addiu   sp,sp,-32
 4006a8: 279c8930 addiu   gp,gp,-30416
 4006ac: afbf001c sw      ra,28(sp)
 4006b0: afb00018 sw      s0,24(sp)
 4006b4: afbc0010 sw      gp,16(sp)
 4006b8: 8f998034 lw      t9,-32716(gp)
 4006bc: 3c040040 lui      a0,0x40
 4006c0: 0320f809 jalr     t9
 4006c4: 248408d0 addiu   a0,a0,2256
 4006c8: 8fbc0010 lw      gp,16(sp)
; アドレスxの高ビットを準備する:
 4006cc: 3c100041 lui      s0,0x41
 4006d0: 8f998038 lw      t9,-32712(gp)
 4006d4: 3c040040 lui      a0,0x40
; アドレスxの低ビットに加える:
 4006d8: 26050920 addiu   a1,s0,2336
; アドレスxは $a1にあります
 4006dc: 0320f809 jalr     t9
 4006e0: 248408dc addiu   a0,a0,2268
 4006e4: 8fbc0010 lw      gp,16(sp)
; アドレスxの高ビットは $s0にあります:
; アドレスxの低ビットに加えて、メモリからwordをロードする:
 4006e8: 8e050920 lw      a1,2336(s0)
; xの値は $a1にあります
 4006ec: 8f99803c lw      t9,-32708(gp)
 4006f0: 3c040040 lui      a0,0x40
 4006f4: 0320f809 jalr     t9
 4006f8: 248408e0 addiu   a0,a0,2272
 4006fc: 8fbc001c lw      ra,28(sp)
```

```

400700: 00001021  move    v0,zero
400704: 8fb00018  lw      s0,24(sp)
400708: 03e00008  jr      ra
40070c: 27bd0020  addiu   sp,sp,32

```

アドレスは LUI と ADDIU を使用して形成されていますが、アドレスの上位部分はまだ \$S0 レジスタにあり、LW (「Load Word」) 命令でオフセットをエンコードすることができます。変数から値をロードして printf() に渡すには十分です。

一時的なデータを保持するレジスタの先頭にはTが付いていますが、ここでは接頭辞Sが付いています。その内容は他の関数で使用する前に保持しておく必要があります。

そのため、\$S0 の値は0x4006ccのアドレスに設定されており、scanf() 呼び出し後に0x4006e8番地で再び使用されています。scanf() 関数は値を変更しません。

第1.9.4節scanf()

前述のように、今日 scanf() を使用するのはいちと古めかしいです。しかし、必要ならば、scanf() がエラーなく正しく終了するかどうかを確認する必要があります。

```

#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    if (scanf ("%d", &x)==1)
        printf ("You entered %d...\n", x);
    else
        printf ("What you entered? Huh?\n");

    return 0;
};

```

標準では、scanf()⁷⁴関数は正常に読み取られたフィールドの数を返します。

私たちの場合、すべてがうまく行き、ユーザーが数字を入力した場合、scanf() は1を返し、エラー（またはEOF⁷⁵）では0を返します。

scanf() の戻り値をチェックするためのCコードを追加し、エラーの場合にはエラーメッセージを出力してみましょう。

期待どおりに動作します。

```

C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe

```

⁷⁴scanf, wscanf: [MSDN](#)

⁷⁵End of File (ファイル終端)

```
Enter X:
ouch
What you entered? Huh?
```

MSVC: x86

アセンブリ出力 (MSVC 2010) の内容は次のとおりです。

```

        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

caller 関数 (main()) は **callee** 関数 (scanf()) の結果を必要とするため、呼び出し先は EAX レジスタに返します。

我々は、CMP EAX, 1 (CoMPare) の指示によりそれをチェックします。つまり、EAX レジスタの値と1を比較します。

JNE 条件ジャンプが CMP 命令の後に続きます。JNE は *Jump if Not Equal* の略です。

したがって、EAX レジスタの値が1に等しくない場合、CPUは JNE オペランドに記述されているアドレス (この場合は \$LN2@main) に実行を渡します。このアドレスに制御を渡すと、CPUは printf() を引数 What you entered? Huh? で実行します。しかし、すべてがうまくいけば、条件付きジャンプは取られず、別の printf() 呼び出しが 'You entered %d...' と x の値を引数にとって実行されます。

この場合、2番目の printf() は実行されないため、その前に JMP があります (無条件ジャンプ)。2番目の printf() の後、戻り値0を実装する XOR EAX, EAX 命令の直前に制御を渡します。

したがって、ある値を別の値と比較することは、通常、CMP/Jcc 命令ペアによって実装されると言えます cc は条件コードです。CMP は2つの値を比較し、プロセッサフラグ⁷⁶を設定します。Jcc はこれらのフラグをチェックし、指定されたアドレスに制御を渡すかどうかを決定します。

⁷⁶x86フラグは以下を参照: wikipedia

これは逆説的に聞こえるかもしれませんが、CMP 命令は実際には SUB（減算）です。すべての算術命令は、CMP だけでなくプロセッサフラグを設定します。1と1を比較し、1-1が0であるため、ZFフラグが設定されます（最後の結果が0であることを意味します）。オペランドが等しい場合を除いて、ZF は設定できません。JNE は ZF フラグのみをチェックし、設定されていない場合にジャンプします。JNEは実際にはJNZ (*Jump if Not Zero*) の同義語です。アセンブラは、JNE命令とJNZ命令の両方を同じオペコードに変換します。したがって、CMP 命令は SUB 命令で置き換えることができ、SUB が最初のオペランドの値を変更するという違いを除けば、ほとんどすべてが問題ありません。CMP は結果を保存しない SUB ですが、フラグに影響します。

MSVC: x86: IDA

IDAを実行してIDAを実行しようとしています。ところで、初心者の方は、MSVCで/MD オプションを使用することをお勧めします。つまり、これらの標準関数はすべて実行可能ファイルにリンクされず、代わりに MSVCR*.DLL ファイルからインポートされます。したがって、どの標準関数が使用され、どこでどこが使用されているのかが分かりやすくなります。

IDA のコードを分析する際には、自分自身（と他者）のためにノートを残すことが非常に役に立ちます。例えば、この例を分析すると、エラーが発生した場合に JNZ がトリガーされることがわかります。カーソルをラベルに移動して「n」を押し、「エラー」に名前を変更することができます。別のラベルを作成し、「終了」にします。以下が私の環境での結果です。

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
.text:00401000 argc = dword ptr 8
.text:00401000 argv = dword ptr 0Ch
.text:00401000 envp = dword ptr 10h
.text:00401000
.text:00401000      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      push    ecx
.text:00401004      push    offset Format ; "Enter X:\n"
.text:00401009      call    ds:printf
.text:0040100F      add     esp, 4
.text:00401012      lea     eax, [ebp+var_4]
.text:00401015      push    eax
.text:00401016      push    offset aD ; "%d"
.text:0040101B      call    ds:scanf
.text:00401021      add     esp, 8
.text:00401024      cmp     eax, 1
.text:00401027      jnz     short error
.text:00401029      mov     ecx, [ebp+var_4]
.text:0040102C      push    ecx
.text:0040102D      push    offset aYou ; "You entered %d...\n"
.text:00401032      call    ds:printf
.text:00401038      add     esp, 8
.text:0040103B      jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D      push    offset aWhat ; "What you entered? Huh?\n"
```



```
.text:00401042      call     ds:printf
.text:00401048      add      esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B      xor      eax, eax
.text:0040104D      mov      esp, ebp
.text:0040104F      pop      ebp
.text:00401050      retn
.text:00401050 _main endp
```

これで、コードを少し理解しやすくなりました。しかし、すべての命令についてコメントするのは良い考えではありません。

また、IDA の関数の一部を隠すこともできます。ブロックをマークするには、Ctrl-「-」を数値パッドに入力し、代わりに表示するテキストを入力します。

2つのブロックを隠して名前を付けましょう。

```
.text:00401000 _text segment para public 'CODE' use32
.text:00401000      assume cs:_text
.text:00401000      ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024      cmp     eax, 1
.text:00401027      jnz     short error
.text:00401029 ; print result
.text:0040103B      jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D      push    offset aWhat ; "What you entered? Huh?\n"
.text:00401042      call     ds:printf
.text:00401048      add      esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B      xor      eax, eax
.text:0040104D      mov      esp, ebp
.text:0040104F      pop      ebp
.text:00401050      retn
.text:00401050 _main endp
```

以前に折りたたまれた部分を展開するには、数値パッドでCtrl-「+」を使用します。

「スペース」を押すと、IDA が関数をグラフとして表示するのを見ることができます。

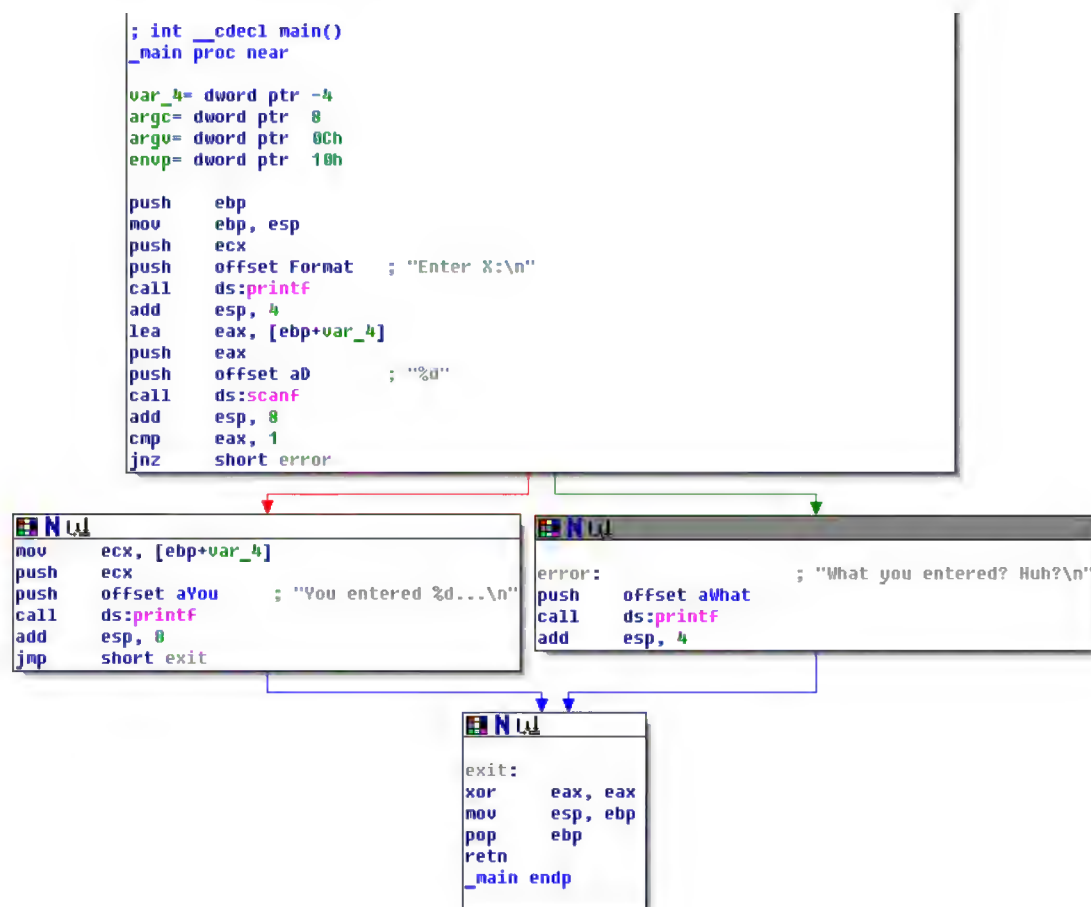


図 1.18: Graph mode in IDA

各条件ジャンプの後、緑と赤の2つの矢印があります。緑の矢印は、ジャンプがトリガされた場合に実行されるブロックを指し、そうでない場合は赤を指します。

このモードでノードを折りたたみ、名前を付けることもできます ([q グループノード)。3つのブロックでやってみましょう。

```
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
```

```
push    ebp
mov     ebp, esp
push    ecx
push    offset Format    ; "Enter X:\n"
call    ds:printf
add     esp, 4
lea     eax, [ebp+var_4]
push    eax
push    offset aD        ; "%d"
call    ds:scanf
add     esp, 8
cmp     eax, 1
jnz     short error
```

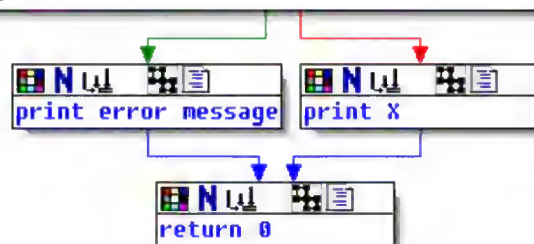


図 1.19: Graph mode in IDA with 3 nodes folded

それは非常に便利です。リバースエンジニアの仕事（および他の研究者の仕事）の非常に重要な部分は、彼らが扱う情報の量を減らすことであると言えます。

MSVC: x86 + OllyDbg

OllyDbg でプログラムをハックしようとして、scanf() が常にエラーなく動作するようにしましょう。ローカル変数のアドレスが scanf() に渡されると、変数には最初にいくつかのランダムなガベージが含まれます。この場合、0x6E494714 です。

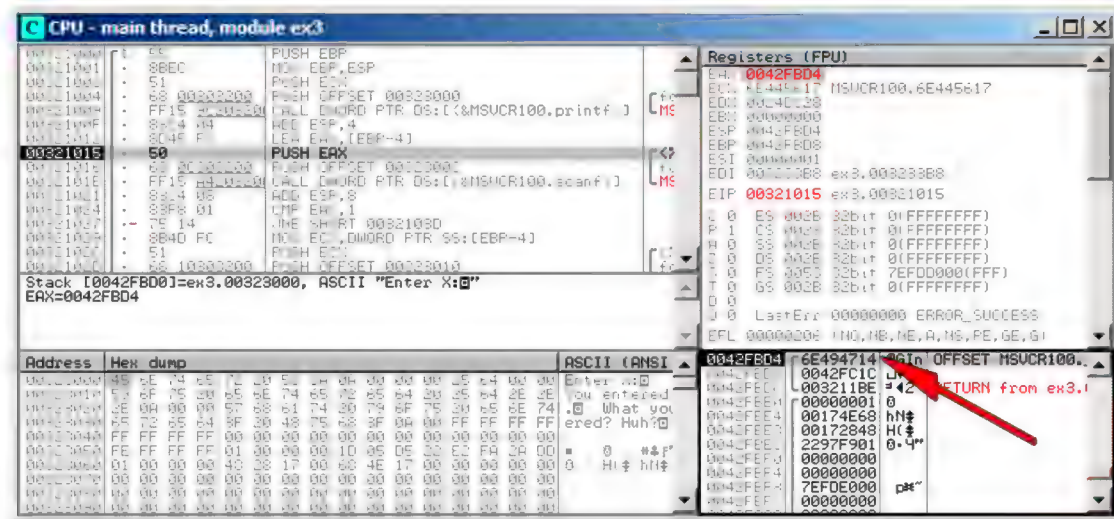


図 1.20: OllyDbg: passing variable address into scanf()

`scanf()` が実行されている間、コンソールでは、「asdasd」のように、数字ではないものを入力します。`scanf()` は、エラーが発生したことを示す `EAX` が0で終了します。

また、スタック内のローカル変数をチェックし、変更されていないことに注意してください。実際、`scanf()` は何を書いていますか？ゼロを返す以外は何もしませんでした。

私たちのプログラムを「ハックする」ようにしましょう。`EAX` を右クリックし、オプションの中に「Set to 1」があります。これが必要なものです。

`EAX` には1があるので、以下のチェックを意図どおりに実行し、`printf()` は変数の値をスタックに出力します。

プログラム (F9) を実行すると、コンソールウィンドウで次のように表示されます。

Listing 1.80: console window

```
Enter X:
asdasd
You entered 1850296084...
```

実際、1850296084はスタック (0x6E494714) の数値を10進表現したものです！

MSVC: x86 + Hiew

これは、実行可能ファイルのパッチ適用の簡単な例としても使用できます。実行可能ファイルにパッチを適用して、入力内容にかかわらずプログラムが常に入力を出力するようにすることがあります。

実行可能ファイルが外部の MSVCR*.DLL (つまり/MD オプション付き)⁷⁷ に対してコンパイルされていると仮定すると、.text セクションの先頭に main() 関数があります。Hiewで実行可能ファイルを開き、.text セクションの先頭を見つけましょう (Enter、F8、F6、Enter、Enter)。

以下のように見えます。

```

Hiew: ex3.exe
C:\Polygon\ollydbg\ex3.exe  BFR0 ----- a32 PE .00401000 Hiew
.00401000: 55          push     ebp
.00401001: 8BEC       mov      ebp,esp
.00401003: 51         push     ecx
.00401004: 6800304000 push     000403000 ;'Enter X:' --E1
.00401009: FF1594204000 call     printf
.0040100F: 83C404     add      esp,4
.00401012: 8D45FC     lea      eax,[ebp][-4]
.00401015: 50         push     eax
.00401016: 6800304000 push     00040300C --E2
.0040101B: FF159C204000 call     scanf
.00401021: 83C408     add      esp,8
.00401024: 83F801     cmp      eax,1
.00401027: 7514     jnz      .00040103D --E3
.00401029: 8B4DFC     mov      ecx,[ebp][-4]
.0040102C: 51         push     ecx
.0040102D: 6810304000 push     000403010 ;'You entered %d...'
.00401032: FF1594204000 call     printf
.00401038: 83C408     add      esp,8
.0040103B: EB0E     jmps     .00040104B --E5
.0040103D: 6820304000 push     000403024 ;'What you entered?'
.00401042: FF1594204000 call     printf
.00401048: 83C404     add      esp,4
.0040104B: 33C0     xor      eax,eax
.0040104D: 8BE5     mov      esp,ebp
.0040104F: 5D         pop      ebp
.00401050: C3         ret     ; ~~~~~
.00401051: B84D5A0000 mov      eax,000005A4D ;' ZM'
1Global 2Local 3Cryba 4Reload 5OrdLdr 6String 7Direct 8Table 9byte 10Leave 11Nat

```

図 1.21: Hiew: main() function

HiewはASCIIZ⁷⁸文字列を検索し、インポートされた関数の名前と同様に表示します。

⁷⁷ 「ダイナミックリンク」とも呼ばれる

⁷⁸ ASCII Zero (ヌル終端文字列)

カーソルを 00401027 番地（ここでバイパスする JNZ 命令がある場所）に移動し、F3を押し、「9090」（2つのNOPを意味する）と入力します。

図 1.22: Hiew: replacing JNZ with two NOPs

その後、F9（更新）を押します。これで、実行可能ファイルがディスクに保存されます。私たちが望むように動作します。

2つのNOPはおそらく最も美しいアプローチではありません。この命令をパッチする別の方法は、第2オペコードバイトに0を書き込むことであり（[jump offset](#)）、JNZ は常に次の命令にジャンプします。

また、最初のバイトを EB で置き換え、2番目のバイト（[jump offset](#)）には触れないでください。私たちは常に無条件のジャンプを得るでしょう。この場合、エラーメッセージは入力に関係なく毎回表示されます。

MSVC: x64

ここではx86-64の32ビットである *int* 型変数について説明しているので、ここではレジスタの32ビット部分（E-を前に付ける）も同様に使用されています。ただし、ポインタを使用している間は、64ビットのレジスタ部分が使用され、先頭に R-が付きます。

Listing 1.81: MSVC 2012 x64

```

_DATA    SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA    ENDS

_TEXT    SEGMENT
x$ = 32
main     PROC
$LN5:
    sub     rsp, 56
    lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call    printf
    lea     rdx, QWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
    call    scanf
    cmp     eax, 1
    jne     SHORT $LN2@main
    mov     edx, DWORD PTR x$[rsp]
    lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
    call    printf
    jmp     SHORT $LN1@main
$LN2@main:
    lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
    call    printf
$LN1@main:
    ; 0をリターン
    xor     eax, eax
    add     rsp, 56
    ret     0
main     ENDP
_TEXT    ENDS
END

```

ARM**ARM: 最適化 Keil 6/2013 (Thumbモード)**

Listing 1.82: 最適化 Keil 6/2013 (Thumbモード)

```

var_8    = -8

        PUSH    {R3,LR}
        ADR     R0, aEnterX      ; "Enter X:\n"

```

```

        BL      __2printf
        MOV     R1, SP
        ADR     R0, aD          ; "%d"
        BL      __0scanf
        CMP     R0, #1
        BEQ     loc_1E
        ADR     R0, aWhatYouEntered ; "What you entered? Huh?\n"
        BL      __2printf

loc_1A:                                ; CODE XREF: main+26
        MOV     R0, #0
        POP     {R3,PC}

loc_1E:                                ; CODE XREF: main+12
        LDR     R1, [SP,#8+var_8]
        ADR     R0, aYouEnteredD____ ; "You entered %d...\n"
        BL      __2printf
        B       loc_1A

```

ここでの新しい命令は CMP と [BEQ⁷⁹](#) です。

CMP は同じ名前の x86 命令に似ていますが、他の引数から引数の1つを減算し、必要に応じて条件フラグを更新します。

オペランドが互いに等しい場合、または最後の計算の結果が0の場合、またはZフラグが1の場合、[BEQ](#) は別のアドレスにジャンプします。これは x86 では JZ として動作します。

それ以外はすべてシンプルです。実行フローが2つの分岐に分岐した後、関数の戻り値として0が R0 に書き込まれた時点で分岐が収束し、関数が終了します。

ARM64

Listing 1.83: 非最適化 GCC 4.9.1 ARM64

```

1  .LC0:
2      .string "Enter X:"
3  .LC1:
4      .string "%d"
5  .LC2:
6      .string "You entered %d...\n"
7  .LC3:
8      .string "What you entered? Huh?"
9  f6:
10 ; スタックフレームにFPとLRと保存
11     stp     x29, x30, [sp, -32]!
12 ; スタックフレームを設定 (FP=SP)
13     add     x29, sp, 0
14 ; "Enter X:" 文字列へのポインタをロード
15     adrp    x0, .LC0
16     add     x0, x0, :lo12: .LC0
17     bl      puts

```

⁷⁹(PowerPC, ARM) Branch if Equal

```

18 ; "%d" 文字列へのポインタをロード
19     adrp    x0, .LC1
20     add     x0, x0, :lo12:.LC1
21 ; ローカルスタックにある変数xのアドレスを計算
22     add     x1, x29, 28
23     bl      isoc99_scanf
24 ; W0にscanf() の戻り値が入っている
25 ; チェックする
26     cmp     w0, 1
27 ; BNEはイコールでない場合に分岐する
28 ; だから、W0<=1の場合、L2にジャンプする
29     bne     .L2
30 ; W0=1の場合、エラーなし
31 ; ローカルスタックからxの値をロードする
32     ldr     w1, [x29,28]
33 ; "You entered %d...\n" 文字列へのポインタをロードする
34     adrp    x0, .LC2
35     add     x0, x0, :lo12:.LC2
36     bl      printf
37 ; "What you entered? Huh?" 文字列を表示するコードをスキップする
38     b       .L3
39 .L2:
40 ; "What you entered? Huh?" 文字列へのポインタをロードする
41     adrp    x0, .LC3
42     add     x0, x0, :lo12:.LC3
43     bl      puts
44 .L3:
45 ; 0をリターン
46     mov     w0, 0
47 ; FPとLRを元に戻す:
48     ldp     x29, x30, [sp], 32
49     ret

```

この場合のコードフローは、CMP/BNE（Branch if Not Equal）命令のペアを使用して分岐します。

MIPS

Listing 1.84: 最適化 GCC 4.4.5 (IDA)

```

.text:004006A0 main:
.text:004006A0
.text:004006A0 var_18    = -0x18
.text:004006A0 var_10    = -0x10
.text:004006A0 var_4     = -4
.text:004006A0
.text:004006A0          lui     $gp, 0x42
.text:004006A4          addiu   $sp, -0x28
.text:004006A8          li      $gp, 0x418960
.text:004006AC          sw      $ra, 0x28+var_4($sp)
.text:004006B0          sw      $gp, 0x28+var_18($sp)
.text:004006B4          la      $t9, puts
.text:004006B8          lui     $a0, 0x40

```

```

.text:004006BC    jalr    $t9 ; puts
.text:004006C0    la      $a0, aEnterX      # "Enter X:"
.text:004006C4    lw      $gp, 0x28+var_18($sp)
.text:004006C8    lui     $a0, 0x40
.text:004006CC    la      $t9, __isoc99_scanf
.text:004006D0    la      $a0, aD           # "%d"
.text:004006D4    jalr    $t9 ; __isoc99_scanf
.text:004006D8    addiu   $a1, $sp, 0x28+var_10 # branch delay slot
.text:004006DC    li      $v1, 1
.text:004006E0    lw      $gp, 0x28+var_18($sp)
.text:004006E4    beq     $v0, $v1, loc_40070C
.text:004006E8    or      $at, $zero        # branch delay slot, NOP
.text:004006EC    la      $t9, puts
.text:004006F0    lui     $a0, 0x40
.text:004006F4    jalr    $t9 ; puts
.text:004006F8    la      $a0, aWhatYouEntered # "What you entered?
                        Huh?"
.text:004006FC    lw      $ra, 0x28+var_4($sp)
.text:00400700    move    $v0, $zero
.text:00400704    jr      $ra
.text:00400708    addiu   $sp, 0x28

.text:0040070C    loc_40070C:
.text:0040070C    la      $t9, printf
.text:00400710    lw      $a1, 0x28+var_10($sp)
.text:00400714    lui     $a0, 0x40
.text:00400718    jalr    $t9 ; printf
.text:0040071C    la      $a0, aYouEnteredD___ # "You entered
                        %d...\n"
.text:00400720    lw      $ra, 0x28+var_4($sp)
.text:00400724    move    $v0, $zero
.text:00400728    jr      $ra
.text:0040072C    addiu   $sp, 0x28

```

scanf() は、その作業の結果をレジスタ \$V0 に返します。アドレス0x004006E4は、\$V0 の値と \$V1 (1は \$V1 以前の0x004006DCに格納されています) と比較することでチェックされます。BEQ は「Branch Equal」の略です。2つの値が等しい場合 (すなわち、成功した場合)、アドレス0x0040070Cにジャンプします。

練習問題

見てきたように、JNE/JNZ 命令は JE/JZ 命令に簡単に置き換えることができます (BNE by BEQ またはその逆)。しかし、基本ブロックも入れ替える必要があります。いくつかの例でこれを試してください。

第1.9.5節練習問題

- <http://challenges.re/53>

第1.10節渡された引数にアクセスする

さて、**caller**関数が引数を**callee**側にスタック経由で渡していることが分かりました。しかし、**callee**関数はどうやって引数にアクセスするのでしょうか？

Listing 1.85: simple example

```
#include <stdio.h>

int f (int a, int b, int c)
{
    return a*b*c;
};

int main()
{
    printf ("%d\n", f(1, 2, 3));
    return 0;
};
```

第1.10.1節x86

MSVC

コンパイルして得られるものを次に示します（MSVC 2010 Express）。

Listing 1.86: MSVC 2010 Express

```
_TEXT    SEGMENT
_a$ = 8      ; size = 4
_b$ = 12     ; size = 4
_c$ = 16     ; size = 4
_f        PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    imul    eax, DWORD PTR _b$[ebp]
    add     eax, DWORD PTR _c$[ebp]
    pop     ebp
    ret     0
_f        ENDP

_main     PROC
    push    ebp
    mov     ebp, esp
    push    3 ; 3番目の引数
    push    2 ; 2番目の引数
    push    1 ; 1番目の引数
    call    _f
    add     esp, 12
    push    eax
    push    OFFSET $SG2463 ; '%d', 0aH, 00H
    call    _printf
```

```

    add     esp, 8
    ; 0をリターン
    xor     eax, eax
    pop     ebp
    ret     0
_main     ENDP

```

main() 関数は3つの数値をスタックにプッシュし、f(int,int,int) を呼び出すことがわかります。

f() 内の引数アクセスは、ローカル変数と同じ方法で `_a$ = 8` のようなマクロの助けを借りて構成されますが、正のオフセット（プラスで扱われます）を持ちます。したがって、`_a$` マクロを EBP レジスタの値に追加することによって [stack frame](#) の外側を処理しています。

次に、`a` の値が EAX に格納されます。IMUL 命令実行後、EAX の値は EAX の値と `_b` の内容の積です。

その後、ADD は `_c` の値を EAX に追加します。

EAX の値は移動する必要はありません。すでに存在している必要があります。[caller](#)に戻ると、EAX 値をとり、printf() の引数として使用します。

MSVC + OllyDbg

これを OllyDbg で説明しましょう。最初の引数（最初の引数）を使用する f() の最初の命令をトレースすると、EBP が赤い四角でマークされた[stack frame](#)を指していることがわかります。

[stack frame](#)の最初の要素は EBP のセーブされた値であり、2番目の要素はRAであり、3番目の要素は最初の関数の引数であり、2番目と3番目の要素です。

最初の関数引数にアクセスするには、EBP にちょうど8（2つの32ビットワード）を追加する必要があります。

OllyDbg はこれを知っているので、

「RETURN from」や「Arg1 = ...」などのスタック要素にコメントを追加しました。

注意：関数の引数は、関数のスタックフレームのメンバーではなく、むしろ[caller](#)関数のスタックフレームのメンバーです。

したがって、OllyDbg は別のスタックフレームのメンバーとして「Arg」要素をマークしました。

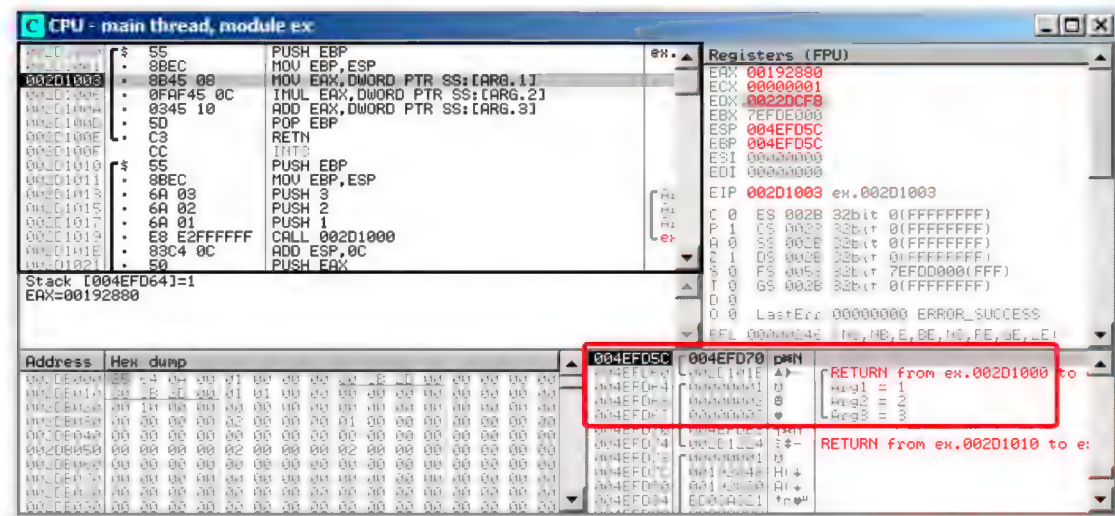


図 1.23: OllyDbg: inside of f() function

GCC

GCC 4.4.1で同じものをコンパイルし、IDA の結果を見てみましょう。

Listing 1.87: GCC 4.4.1

```

f      public f
f      proc near

arg_0  = dword ptr 8
arg_4  = dword ptr 0Ch
arg_8  = dword ptr 10h

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0] ; 1番目の引数
        imul    eax, [ebp+arg_4] ; 2番目の引数
        add     eax, [ebp+arg_8] ; 3番目の引数
        pop     ebp
        retn

f      endp

main   public main
main   proc near

var_10 = dword ptr -10h
var_C  = dword ptr -0Ch
var_8  = dword ptr -8

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFF0h

```



```

    sub     esp, 10h
    mov     [esp+10h+var_8], 3 ; 3番目の引数
    mov     [esp+10h+var_C], 2 ; 2番目の引数
    mov     [esp+10h+var_10], 1 ; 1番目の引数
    call    f
    mov     edx, offset aD ; "%d\n"
    mov     [esp+10h+var_C], eax
    mov     [esp+10h+var_10], edx
    call    _printf
    mov     eax, 0
    leave
    retn
main      endp

```

結果はほぼ同じで、以前に説明したいくつかの小さな違いがあります。

スタックポインタは2つの関数呼び出し (fとprintf) の後にセットバックされません。最後から2番目の **LEAVE** 命令 (?? on page ??) 命令が最後にこれを処理するためです。

第1.10.2節x64

この話はx86-64では少し違っています。関数の引数 (最初の4つまたは最初の6つ) はレジスタに渡されます。つまり、**callee**はレジスタからレジスタを読み込みます。

MSVC

最適化 MSVC:

Listing 1.88: 最適化 MSVC 2012 x64

```

$SG2997 DB      '%d', 0aH, 00H

main      PROC
    sub     rsp, 40
    mov     edx, 2
    lea     r8d, QWORD PTR [rdx+1] ; R8D=3
    lea     ecx, QWORD PTR [rdx-1] ; ECX=1
    call    f
    lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
    mov     edx, eax
    call    printf
    xor     eax, eax
    add     rsp, 40
    ret     0
main      ENDP

f         PROC
; ECX - 1番目の引数
; EDX - 2番目の引数
; R8D - 3番目の引数
    imul    ecx, edx
    lea     eax, DWORD PTR [r8+rcx]
    ret     0

```

f ENDP

見てわかるように、コンパクトな関数 `f()` はすべての引数をレジスタから取ります。

ここでの `LEA` 命令は加算に使用され、明らかにコンパイラは `ADD` よりも速いと考えました。

`LEA` は、第1および第3の `f()` 引数を準備するために `main()` 関数でも使用されます。コンパイラは、`MOV` 命令を使用してレジスタに値をロードする通常の方法よりも速く動作すると判断する必要があります。

非最適化MSVCの出力を見てみましょう。

Listing 1.89: MSVC 2012 x64

```
f                proc near
; シャドースペース
arg_0           = dword ptr 8
arg_8           = dword ptr 10h
arg_10          = dword ptr 18h

                ; ECX - 1番目の引数
                ; EDX - 2番目の引数
                ; R8D - 3番目の引数
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f                endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; 3番目の引数
                mov     edx, 2 ; 2番目の引数
                mov     ecx, 1 ; 1番目の引数
                call    f
                mov     edx, eax
                lea     rcx, $SG2931 ; "%d\n"
                call    printf

                ; 0をリターン
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

レジスタからの3つの引数は何らかの理由でスタックに保存されるため、ややこしいことになっています。

これは「シャドウスペース」と呼ばれます。⁸⁰ すべてのWin64は、そこにある4つのレジスタ値をすべて保存することができます（必須ではありません）。これは2つの理由で行われます。1) 入力引数にレジスタ全体（または4つのレジスタ）を割り当てるのはあまりにも贅沢なので、スタック経由でアクセスされます。2) デバッガはブレークで関数の引数をどこに見つけるか常に認識しています。⁸¹

だから、大規模な関数の中には、実行中にそれらを使用したい場合、入力引数を「シャドウスペース」に保存することができますが、私たちのような小さな関数ではそうでないかもしれません。

スタックに「シャドウスペース」を割り当てるのはcallerの責任です。

GCC

最適化 GCCはまあまあわかりやすいコードを生成します。

Listing 1.90: 最適化 GCC 4.4.6 x64

```
f:
    ; EDI - 1番目の引数
    ; ESI - 2番目の引数
    ; EDX - 3番目の引数
    imul    esi, edi
    lea     eax, [rdx+rsi]
    ret

main:
    sub     rsp, 8
    mov     edx, 3
    mov     esi, 2
    mov     edi, 1
    call    f
    mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
    mov     esi, eax
    xor     eax, eax ; 渡されたベクトルレジスタの数
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret
```

非最適化 GCC:

Listing 1.91: GCC 4.4.6 x64

```
f:
    ; EDI - 1番目の引数
    ; ESI - 2番目の引数
    ; EDX - 3番目の引数
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
```

⁸⁰MSDN

⁸¹MSDN

```

        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0 ; 渡されたベクトルレジスタの数
        call    printf
        mov     eax, 0
        leave
        ret

```

System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]⁸²⁾ には「シャドースペース」の要件はありませんが、**callee**はレジスタが不足している場合には引数をどこかに保存します。

GCC: intの代わりにのuint64_t

私たちの例は32ビットintで動作するため、32ビットのレジスタが使用されています (E-が前に付いています)。

64ビット値を使用するためには少し変更する必要があります。

```

#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b*c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                        0x1111111122222222,
                        0x3333333344444444));

    return 0;
}

```

⁸²以下で利用可能 <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

```
};
```

Listing 1.92: 最適化 GCC 4.4.6 x64

```
f      proc near
      imul    rsi, rdi
      lea     rax, [rdx+rsi]
      retn
f      endp

main   proc near
      sub     rsp, 8
      mov     rdx, 3333333344444444h ; 3番目の引数
      mov     rsi, 1111111122222222h ; 2番目の引数
      mov     rdi, 1122334455667788h ; 1番目の引数
      call    f
      mov     edi, offset format ; "%lld\n"
      mov     rsi, rax
      xor     eax, eax ; 渡されたベクトルレジスタの数
      call    _printf
      xor     eax, eax
      add     rsp, 8
      retn
main   endp
```

コードは同じですが、今回はフルサイズのレジスタ（R-が前に付いています）が使用されています。

第1.10.3節ARM

非最適化 Keil 6/2013 (ARMモード)

```
.text:000000A4 00 30 A0 E1      MOV     R3, R0
.text:000000A8 93 21 20 E0      MLA     R0, R3, R1, R2
.text:000000AC 1E FF 2F E1      BX      LR
...
.text:000000B0                main
.text:000000B0 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3      MOV     R2, #3
.text:000000B8 02 10 A0 E3      MOV     R1, #2
.text:000000BC 01 00 A0 E3      MOV     R0, #1
.text:000000C0 F7 FF FF EB      BL      f
.text:000000C4 00 40 A0 E1      MOV     R4, R0
.text:000000C8 04 10 A0 E1      MOV     R1, R4
.text:000000CC 5A 0F 8F E2      ADR     R0, aD_0          ; "%d\n"
.text:000000D0 E3 18 00 EB      BL      __2printf
.text:000000D4 00 00 A0 E3      MOV     R0, #0
.text:000000D8 10 80 BD E8      LDMFD   SP!, {R4,PC}
```

main() 関数は他の2つの関数を呼び出し、3つの値が最初の関数に渡されます（f()）。

前述のように、ARMでは最初の4つの値が通常最初の4つのレジスタ（R0-R3）に渡されます。

f() 関数は、最初の3つのレジスタ (R0-R2) を引数として使用します。

MLA (*Multiply Accumulate*) 命令は最初の2つのオペランド (R3 と R1) を乗算し、3番目のオペランド (R2) を積に加算し、その結果をゼロ関数 (R0) に格納します。

一度に乗算と加算を同時に行うの (*Fused multiply-add*) は非常に便利な操作です。ところで、SIMD⁸³ にFMA命令が登場する前に、x86にそのような命令はありませんでした。

最初の MOV R3, R0 命令は明らかに冗長です (ここでは単一の MLA 命令を代わりに使用できます)。これは最適化されないコンパイルであるため、コンパイラは最適化していません。

BX 命令は、制御を LRレジスタに格納されているアドレスに戻し、必要に応じてプロセッサモードをThumbからARMに、またはその逆に切り替えます。これは、関数 f() がどのような種類のコード (ARMまたはThumb) から認識されていないため、必要な場合があります。したがって、Thumbコードから呼び出された場合、BX は呼び出し元の関数に制御を戻すだけでなく、プロセッサモードをThumbに切り替えます。ARMコード ([ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition, (2012)A2.3.2]) から関数が呼び出されているかどうかを切り替えます。

最適化 Keil 6/2013 (ARMモード)

.text:00000098		f	
.text:00000098 91 20 20 E0	MLA	R0, R1, R0, R2	
.text:0000009C 1E FF 2F E1	BX	LR	

Keilコンパイラによって完全最適化モード (-O3) でコンパイルされた f() 関数があります。

MOV 命令は最適化され (または縮小され)、MLA はすべての入力レジスタを使用し、結果を R0 に配置します。

最適化 Keil 6/2013 (Thumbモード)

.text:0000005E 48 43	MULS	R0, R1
.text:00000060 80 18	ADDSD	R0, R0, R2
.text:00000062 70 47	BX	LR

Thumbモードでは MLA 命令を使用できないため、コンパイラはこれら2つの演算 (乗算と加算) を別々に実行するコードを生成します。

最初に、MULS 命令は R0 に R1 を掛けて、結果をレジスタ R0に残します。2番目の命令 (ADDSD) は結果と R2 を加算して結果をレジスタ R0に残します。

ARM64

最適化 GCC (Linaro) 4.9

⁸³[wikipedia](#)

ここでのすべては簡単です。MADD は乗算/加算を一緒に行う命令です（既に見た MLA に似ています）。3つの引数はすべて、Xレジスタの32ビット部分に渡されます。実際、引数の型は32ビット *int* です。結果は W0 に返されます。

Listing 1.93: 最適化 GCC (Linaro) 4.9

```
f:
    madd    w0, w0, w1, w2
    ret

main:
; FPとLRをスタックフレームに保存
    stp     x29, x30, [sp, -16]!
    mov     w2, 3
    mov     w1, 2
    add     x29, sp, 0
    mov     w0, 1
    bl      f
    mov     w1, w0
    adrp    x0, .LC7
    add     x0, x0, :lo12:.LC7
    bl      printf
; 0をリターン
    mov     w0, 0
; FPとLRを元に戻す
    ldp     x29, x30, [sp], 16
    ret

.LC7:
    .string "%d\n"
```

すべてのデータ型を64ビット `uint64_t` に拡張してテストしましょう：

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
{
    printf ("%lld\n", f(0x1122334455667788,
                        0x1111111122222222,
                        0x3333333344444444));
    return 0;
};
```

```
f:
    madd    x0, x0, x1, x2
    ret

main:
    mov     x1, 13396
```



```

    adrp    x0, .LC8
    stp     x29, x30, [sp, -16]!
    movk    x1, 0x27d0, lsl 16
    add     x0, x0, :lo12:.LC8
    movk    x1, 0x122, lsl 32
    add     x29, sp, 0
    movk    x1, 0x58be, lsl 48
    bl      printf
    mov     w0, 0
    ldp     x29, x30, [sp], 16
    ret

.LC8:
    .string "%lld\n"

```

f() 関数は同じで、64ビットのXレジスタ全体が使用されます。長い64ビットの値は、レジスタごとにロードされます。これについては、以下で説明します：[1.30.3 on page 537](#)

非最適化 GCC (Linaro) 4.9

非最適化コンパイラはより冗長です。

```

f:
    sub     sp, sp, #16
    str     w0, [sp,12]
    str     w1, [sp,8]
    str     w2, [sp,4]
    ldr     w1, [sp,12]
    ldr     w0, [sp,8]
    mul     w1, w1, w0
    ldr     w0, [sp,4]
    add     w0, w1, w0
    add     sp, sp, 16
    ret

```

コードは、この関数の誰か（または何か）が W0...W2 レジスタを使用する必要がある場合に、その入力引数をローカルスタックに保存します。これにより、元の関数の引数を上書きすることが防止されます。これは、将来必要になる可能性があります。

これは、レジスタセーブエリアと呼ばれます。[*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]⁸⁴。しかし、呼び出し先関数はそれらを保存する義務はありません。これは、「シャドースペース」([1.10.2 on page 126](#)) に多少似ています。

GCC 4.9を最適化すると、なぜこの引数はコードを保存しなくなったのでしょうか？これはいくつかの追加の最適化作業を行い、関数の引数がこの先に必要ではなく、レジスタ W0...W2 が使用されないと結論付けたためです。

また、単一の MADD の代わりに MUL/ADD 命令ペアがあります。

⁸⁴以下で利用可能 http://infocenter.arm.com/help/topic/com.arm.doc.ih00555b/IHI00555B_aapcs64.pdf

第1.10.4節MIPS

Listing 1.94: 最適化 GCC 4.4.5

```

.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000      mult    $a1, $a0
.text:00000004      mflo    $v0
.text:00000008      jr      $ra
.text:0000000C      addu    $v0, $a2, $v0      ; 分岐遅延スロット
; 戻り値は $v0に格納される
.text:00000010 main:
.text:00000010
.text:00000010 var_10 = -0x10
.text:00000010 var_4  = -4
.text:00000010
.text:00000010      lui     $gp, (__gnu_local_gp >> 16)
.text:00000014      addiu   $sp, -0x20
.text:00000018      la      $gp, (__gnu_local_gp & 0xFFFF)
.text:0000001C      sw      $ra, 0x20+var_4($sp)
.text:00000020      sw      $gp, 0x20+var_10($sp)
; cを設定
.text:00000024      li      $a2, 3
; aを設定
.text:00000028      li      $a0, 1
.text:0000002C      jal     f
; bを設定
.text:00000030      li      $a1, 2      ; 分岐遅延スロット
; 結果は $v0にある
.text:00000034      lw      $gp, 0x20+var_10($sp)
.text:00000038      lui     $a0, ($LC0 >> 16)
.text:0000003C      lw      $t9, (printf & 0xFFFF)($gp)
.text:00000040      la      $a0, ($LC0 & 0xFFFF)
.text:00000044      jalr    $t9
; take result of f() 関数の結果を取得しprintf() の2番目の引数に渡す
.text:00000048      move    $a1, $v0      ; 分岐遅延スロット
.text:0000004C      lw      $ra, 0x20+var_4($sp)
.text:00000050      move    $v0, $zero
.text:00000054      jr      $ra
.text:00000058      addiu   $sp, 0x20 ; 分岐遅延スロット

```

最初の4つの関数引数は、A-が前に付いた4つのレジスタに渡されます。

MIPSには、HIとLOの2つの特殊レジスタがあり、MULT 命令の実行中に乗算の64ビット結果が格納されます。

これらのレジスタは、MFL0 および MFHI 命令を使用することによってのみアクセスできます。ここでは MFL0 は乗算結果の低部分を取り、それを \$V0 に格納します。そのため、乗算結果の上位32ビット部分が削除されます (HIレジスタの内容は使用されません)。確かに、ここでは32ビットの *int* データ型を扱います。

最後に、ADDU (「Add Unsigned」) は3番目の引数の値を結果に加えます。

MIPSには、ADD と ADDU の2種類の加算命令があります。それらの違いは、署名性に関連するものではなく、例外に対するものです。ADD では、オーバーフローに関する例外が発生することがあります。これは、たとえばAda [PL](#)で有用⁸⁵であり、サポートされることもあります。ADDU は、オーバーフロー時に例外を発生させません。

C/C++ ではこれをサポートしていないため、この例では ADD の代わりに ADDU が表示されています。

32ビットの結果は \$V0 に残ります。

main() に新しい命令があります (JAL (「Jump and Link」))。

JAL と JALR の違いは、相対オフセットが最初の命令でエンコードされる一方、JALR はレジスタに格納された絶対アドレスにジャンプすることです (「ジャンプレジスタとリンクレジスタ」)。

f() と main() の両方の関数は同じオブジェクトファイルに配置されるので、f() の相対アドレスは既知で固定されています。

第1.11節戻り値を返すことの詳細

x86では、関数の実行結果は通常 EAX レジスタに返されます。⁸⁶ バイトタイプまたは文字 (*char*) の場合は、レジスタ EAX (AL) の最下位部分が使用されます。関数が浮動小数点数を返す場合、代わりにFPUレジスタ ST(0) が使用されます。ARMでは、結果は通常 R0 レジスタに返されます。

第1.11.1節void を返す関数の結果を使ってみる

では、main() 関数の戻り値が *int* 型ではなく *void* 型であると宣言された場合はどうでしょうか？いわゆるスタートアップコードは、以下のように main() を呼び出しています。

```
push envp
push argv
push argc
call main
push eax
call exit
```

言い換えると：

```
exit(main(argc,argv,envp));
```

main() を *void* として宣言すると、明示的に (*return* 文を使って) 何も返されず、main() の最後の EAX レジスタに格納された何かがexit() の唯一の引数になります。おそらく、あなたの関数の実行から放棄されるランダムな値があるでしょう。したがって、プログラムの終了コードは疑似乱数です。

この事実を説明することができます。ここで main() 関数は *void* 戻り値の型を持っていることに注意してください。

⁸⁵<http://blog.regehr.org/archives/1154>

⁸⁶参照: MSDN: Return Values (C++): [MSDN](#)

```
#include <stdio.h>

void main()
{
    printf ("Hello, world!\n");
};
```

Linuxでコンパイルしましょう。

GCC 4.8.1では、printf() を puts() に置き換えました(以下で見てきました：[1.5.3 on page 27](#))、これは printf() のように puts() が出力する文字数を返すので、これは問題ありません。main() が終了する前に EAX がゼロになっていないことに注意してください。

これは、main() の最後の EAX の値に puts() が残した値が含まれていることを意味します。

Listing 1.95: GCC 4.8.1

```
.LC0:
    .string "Hello, world!"
main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0
    call    puts
    leave
    ret
```

終了ステータスを示すbashスクリプトを書いてみましょう。

Listing 1.96: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

実行してみましょう。

```
$ tst.sh
Hello, world!
14
```

14は表示された文字数です。印刷される文字の数は、printf() から EAX/RAX までの「exit code」へのスリップです。

ちなみに、Hex-RaysでC++を逆コンパイルすると、あるクラスのデストラクタで終了する関数に遭遇することがよくあります。

```
...
call    ??1CString@@QAE@XZ ; CString::~ CString(void)
```

```

mov     ecx, [esp+30h+var_C]
pop     edi
pop     ebx
mov     large fs:0, ecx
add     esp, 28h
retn

```

C++標準では、デストラクタは何も返しません、Hex-Raysがそれを知らず、デストラクタとこの関数の両方が *int* を返すと考え、次のような出力が出力されます。

```

...

    return CString::~CString(&Str);
}

```

第1.11.2節関数の戻り値を使わないとどうなる？

`printf()` は正常に出力された文字の数を返しますが、実際にはこの関数の結果はめったに使用されません。

また、値を返す関数を呼び出し、戻り値を使用しないということも可能です。

```

int f()
{
    // 最初の3つのランダム値をスキップする
    rand();
    rand();
    rand();
    // 4番目を使用する
    return rand();
};

```

`rand()` 関数の結果は `EAX` の4つのケースすべてに残されています。

しかし、最初の3つのケースでは、`EAX` の値は使用されていません。

第1.11.3節構造体を返す

戻り値が `EAX` レジスタに残っているという事実に戻りましょう。

そのため、古いCコンパイラでは、1つのレジスタ（通常は *int*）に収まらないものを返す関数を作成することはできませんが、必要なら、関数の引数として渡されたポインタを介して情報を返す必要があります。

したがって、通常、関数が複数の値を返す必要がある場合は、1つだけを返し、残りのすべてのポインタを返します。

構造全体を返すことが可能になっていますが、それはまだあまり一般的ではありません。関数が大きな構造体を返さなければならない場合、呼び出し側はそれを割り当ててポインタを最初の引数を介してプログラマに透過的に渡す必要があります。これは、最初の引数に手動でポインタを渡すのとほぼ同じですが、コンパイラはそれを隠します。

小さな例

```

struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};

```

...以下のコードが出力されます (MSVC 2010 /Ox):

```

$T3853 = 8 ; size = 4
_a$ = 12 ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC ; get_some_values
    mov     ecx, DWORD PTR _a$[esp-4]
    mov     eax, DWORD PTR $T3853[esp-4]
    lea     edx, DWORD PTR [ecx+1]
    mov     DWORD PTR [eax], edx
    lea     edx, DWORD PTR [ecx+2]
    add     ecx, 3
    mov     DWORD PTR [eax+4], edx
    mov     DWORD PTR [eax+8], ecx
    ret     0
?get_some_values@@YA?AUs@@H@Z ENDP ; get_some_values

```

構造体へのポインタの内部渡しのマクロ名は \$T3853 です。

この例は、C99言語拡張を使用して書き直すことができます。

```

struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};

```

Listing 1.97: GCC 4.8.1

```

_get_some_values proc near

```

```

ptr_to_struct    = dword ptr  4
a                = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp

```

この関数は、あたかも構造体へのポインタが渡されたかのように、呼び出し元関数によって割り当てられた構造体のフィールドを埋めるだけです。したがって、パフォーマンス上の欠点はありません。

第1.12節 ポインタ

第1.12.1節 戻り値

ポインタは関数から値を返すためによく使用されます（scanf() 関数の呼び出し（[1.9 on page 84](#)））。

たとえば、関数が2つの値を返す必要がある場合などです。

グローバル変数の例

```

#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
    *sum=x+y;
    *product=x*y;
};

int sum, product;

void main()
{
    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};

```

次のようにコンパイルされます。

Listing 1.98: 最適化 MSVC 2010 (/Ob0)

```

COMM      _product:DWORD

```



```

COMM    _sum:DWORD
$SG2803 DB    'sum=%d, product=%d', 0aH, 00H

_x$ = 8          ; size = 4
_y$ = 12         ; size = 4
_sum$ = 16       ; size = 4
_product$ = 20   ; size = 4
_f1 PROC
    mov     ecx, DWORD PTR _y$(esp-4)
    mov     eax, DWORD PTR _x$(esp-4)
    lea     edx, DWORD PTR [eax+ecx]
    imul    eax, ecx
    mov     ecx, DWORD PTR _product$(esp-4)
    push    esi
    mov     esi, DWORD PTR _sum$(esp)
    mov     DWORD PTR [esi], edx
    mov     DWORD PTR [ecx], eax
    pop     esi
    ret     0
_f1 ENDP

_main PROC
    push    OFFSET _product
    push    OFFSET _sum
    push    456      ; 000001c8H
    push    123     ; 0000007bH
    call    _f1
    mov     eax, DWORD PTR _product
    mov     ecx, DWORD PTR _sum
    push    eax
    push    ecx
    push    OFFSET $SG2803
    call    DWORD PTR __imp__printf
    add     esp, 28
    xor     eax, eax
    ret     0
_main ENDP

```

OllyDbg で見てみましょう。

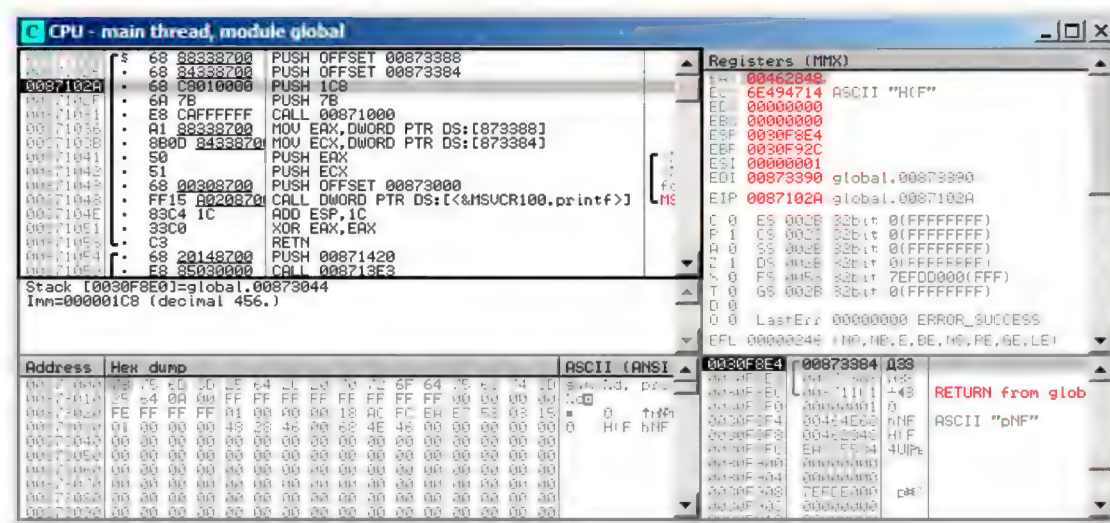
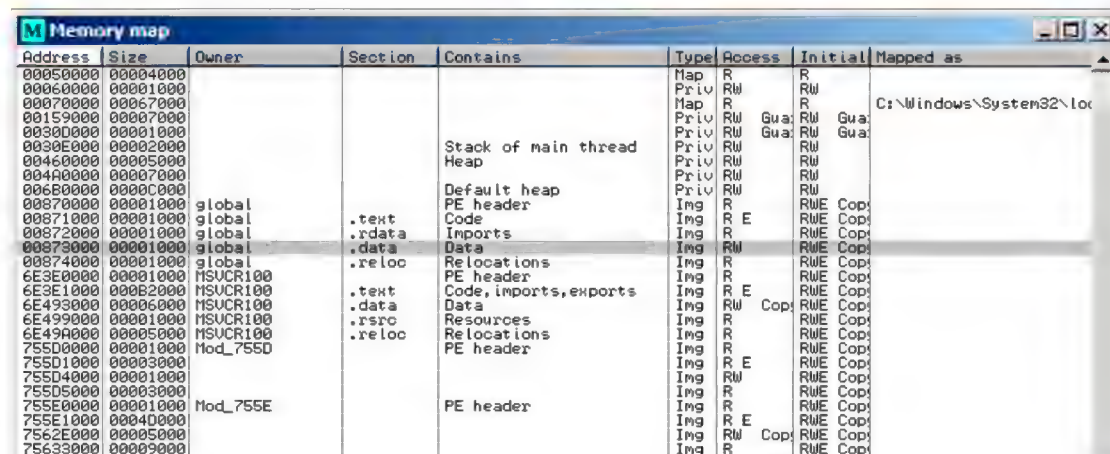


図 1.24: OllyDbg: グローバル変数のアドレスは f1() に渡されます

まず、グローバル変数のアドレスが f1() に渡されます。スタック要素に「Follow in dump」をクリックすると、2つの変数に割り当てられたデータセグメント内の場所を見ることができます。

これらの変数は、初期化されていないデータ（BSSから）が実行開始前にクリアされるため、ゼロにされます。[see ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.8p10]

それらはデータセグメントにあり、Alt-Mを押してメモリマップを確認することで確認できます。



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00050000	00004000				Map	R	R	
00060000	00001000				Priv	RW	RW	
00070000	00007000				Map	R	R	
00159000	00007000				Priv	RW	Gua	RW
0030D000	00001000				Priv	RW	Gua	RW
0030E000	00002000			Stack of main thread	Priv	RW	RW	
00460000	00005000			Heap	Priv	RW	RW	
004A0000	00007000			Default heap	Priv	RW	RW	
006E0000	0000C000	global		PE header	Ing	R	RWE	Cop
00670000	00001000	global	.text	Code	Ing	R	RWE	Cop
00671000	00001000	global	.rdata	Imports	Ing	R	RWE	Cop
00672000	00001000	global	.data	Data	Ing	RW	RWE	Cop
00673000	00001000	global	.reloc	Relocations	Ing	R	RWE	Cop
00674000	00001000	MSUCR100		PE header	Ing	R	RWE	Cop
6E3E0000	00001000	MSUCR100	.text	Code, imports, exports	Ing	R	RWE	Cop
6E3E1000	00002000	MSUCR100	.data	Data	Ing	RW	RWE	Cop
6E493000	00006000	MSUCR100	.rsrc	Resources	Ing	R	RWE	Cop
6E494000	00005000	MSUCR100	.reloc	Relocations	Ing	R	RWE	Cop
755D0000	00001000	Mod_755D		PE header	Ing	R	RWE	Cop
755D1000	00003000				Ing	R	RWE	Cop
755D4000	00001000				Ing	RW	RWE	Cop
755D5000	00003000				Ing	R	RWE	Cop
755E0000	00001000	Mod_755E		PE header	Ing	R	RWE	Cop
755E1000	00004000				Ing	R	RWE	Cop
7562E000	00005000				Ing	RW	RWE	Cop
75633000	00009000				Ing	R	RWE	Cop

図 1.25: OllyDbg: メモリマップ



2つの値がスタック456 (0x1C8) と 123 (0x7B) に表示され、2つのグローバル変数のアドレスも表示されます。

f1() の終わりまでトレースしましょう。左下のウィンドウで、計算結果がグローバル変数にどのように表示されるかを確認します。

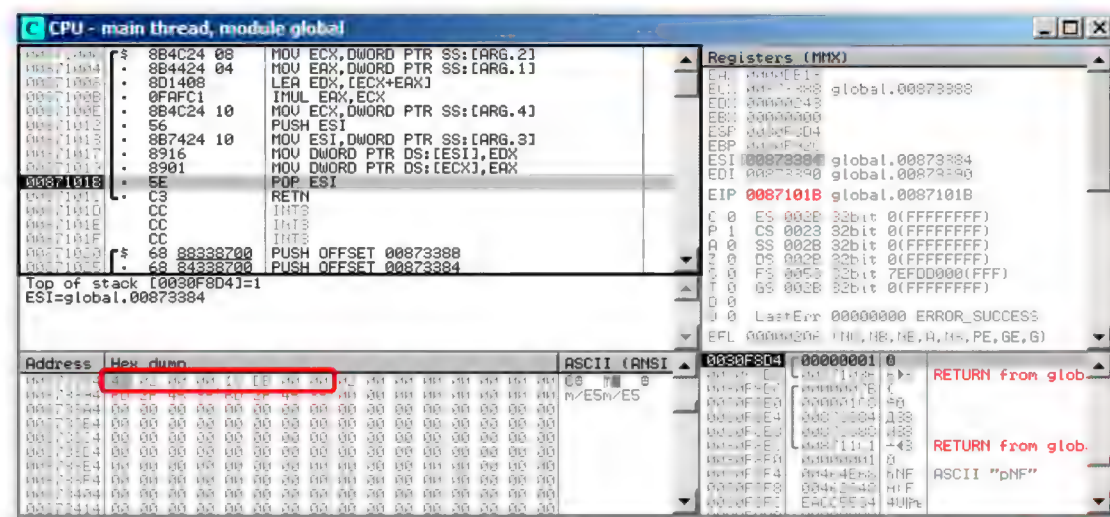


図 1.27: OllyDbg: f1() 実行完了

グローバル変数の値は、(スタックを介して) printf() に渡す準備が整ったレジスタにロードされます。

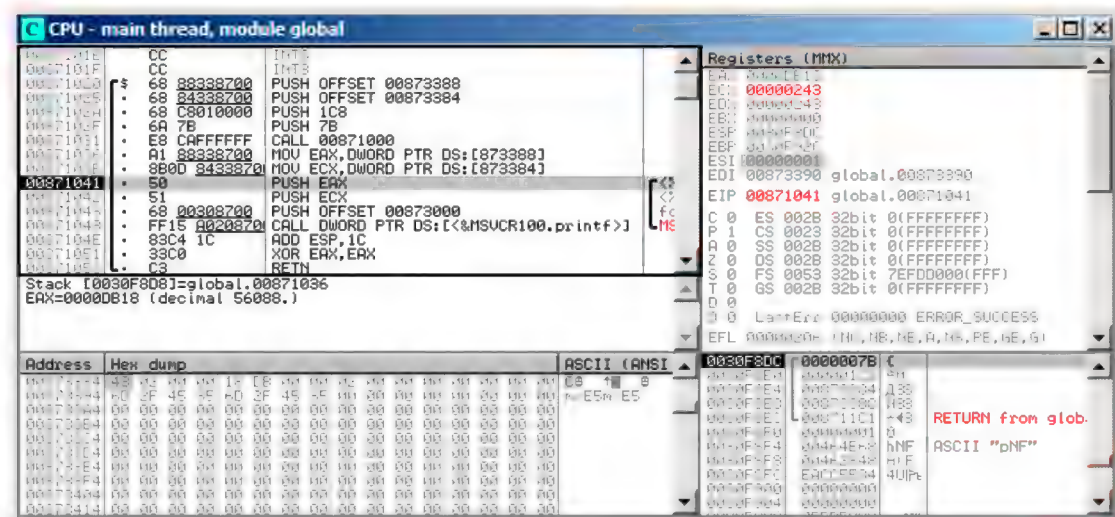


図 1.28: OllyDbg: グローバル変数の値は printf() に渡されます

ローカル変数の例

私たちの例を少し修正しましょう。

Listing 1.99: now the sum and product variables are local

```
void main()
{
    int sum, product; // 変数は関数ローカルにあります

    f1(123, 456, &sum, &product);
    printf ("sum=%d, product=%d\n", sum, product);
};
```

f1() コードは変更されません。main() のコードだけが行います：

Listing 1.100: 最適化 MSVC 2010 (/Ob0)

```
_product$ = -8 ; size = 4
_sum$ = -4 ; size = 4
_main PROC
; Line 10
    sub     esp, 8
; Line 13
    lea     eax, DWORD PTR _product$[esp+8]
    push    eax
    lea     ecx, DWORD PTR _sum$[esp+12]
    push    ecx
    push    456 ; 000001c8H
```

```
    push    123      ; 0000007bH
    call    _f1
; Line 14
    mov     edx, DWORD PTR _product$[esp+24]
    mov     eax, DWORD PTR _sum$[esp+24]
    push    edx
    push    eax
    push    OFFSET $SG2803
    call    DWORD PTR __imp__printf
; Line 15
    xor     eax, eax
    add     esp, 36
    ret     0
```


OllyDbg をもう一度見てみましょう。スタック内のローカル変数のアドレスは 0x2EF854 and 0x2EF858 です。これらがスタックにどのようにプッシュされるのかを確認します。

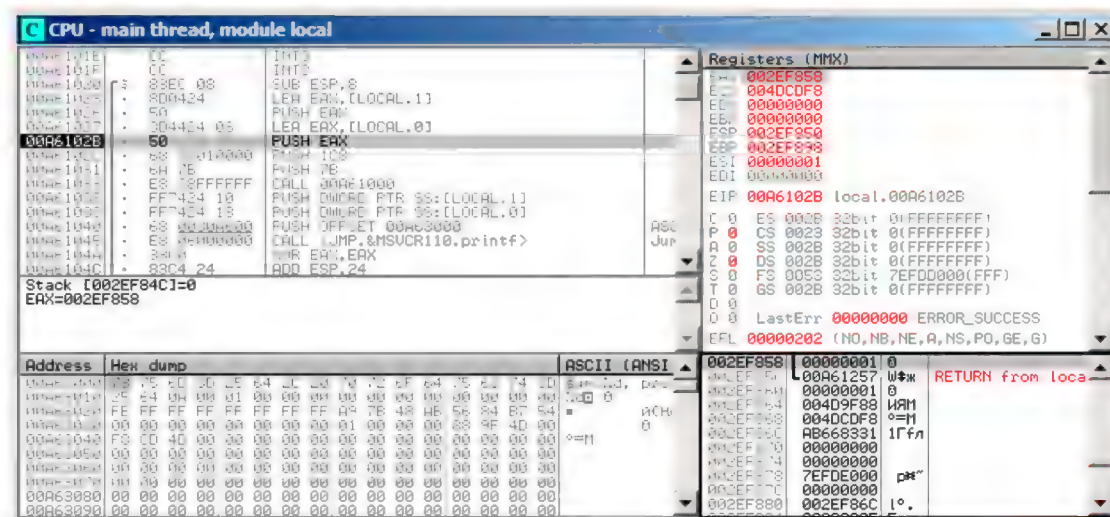


図 1.29: OllyDbg: ローカル変数のアドレスがスタックにプッシュされる

f1() が開始します。今のところ 0x2EF854 and 0x2EF858 のスタックにランダムなゴミだけがあります。

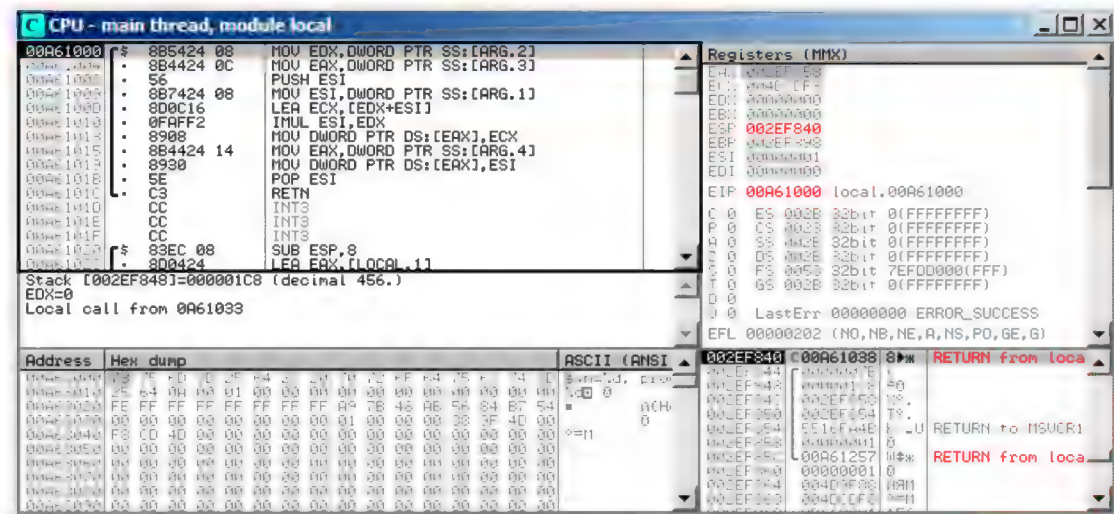


図 1.30: OllyDbg: f1() 開始

f1() が完了。

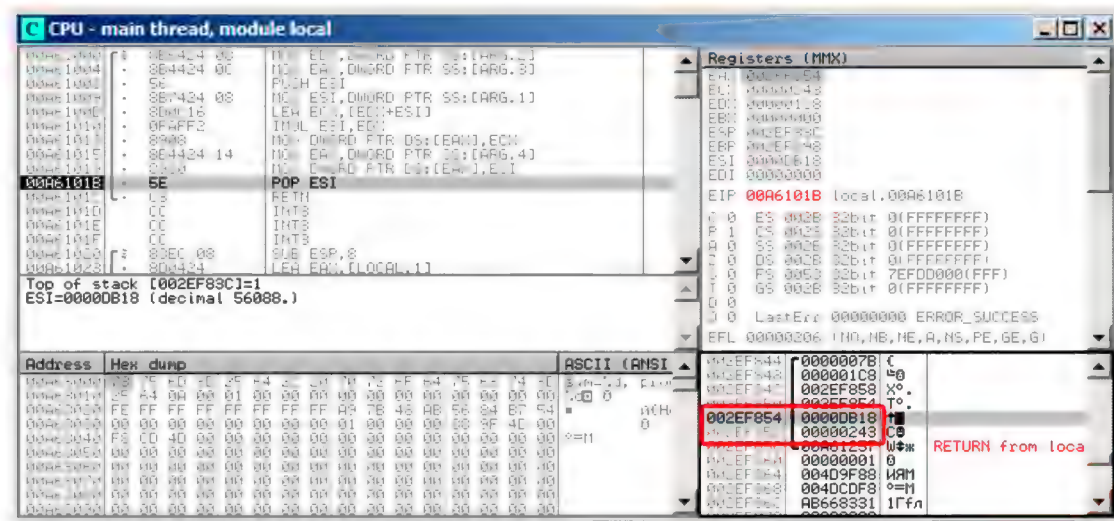


図 1.31: OllyDbg: f1() 実行完了

ここで、アドレス 0x2EF854 and 0x2EF858 に 0xDB18 と 0x243 が見つかります。これらの値は f1() の結果です。

結論

f1() は、メモリ内の任意の場所にポインタを返すことができます。

これは本質的にポインタの有用性です。

ところで、C++ リファレンスはまったく同じように動作します。それらの詳細については、(?? on page ??) を参照してください。

第1.12.2節入力値の入れ替え

こんな仕事をさせてみます

```
#include <memory.h>
#include <stdio.h>

void swap_bytes (unsigned char* first, unsigned char* second)
{
    unsigned char tmp1;
    unsigned char tmp2;

    tmp1=*first;
    tmp2=*second;

    *first=tmp2;
```

```

        *second=tmp1;
};

int main()
{
    // 文字列をヒープにコピーするので、変更することができます
    char *s=strdup("string");

    // 2番目と3番目の文字をスワップする
    swap_bytes (s+1, s+2);

    printf ("%s\n", s);
};

```

見てわかるように、バイトは MOVZX を使用して ECX と EBX の下位8ビット部分にロードされます（これらのレジスタの上位部分がクリアされます）。その後、バイトがスワップバックされます。

Listing 1.101: Optimizing GCC 5.4

```

swap_bytes:
    push    ebx
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+12]
    movzx   ecx, BYTE PTR [edx]
    movzx   ebx, BYTE PTR [eax]
    mov     BYTE PTR [edx], bl
    mov     BYTE PTR [eax], cl
    pop     ebx
    ret

```

両方のバイトのアドレスは引数から取り出され、関数の実行は EDX と EAX にあります。だから私たちはポインタを使用しています。恐らく、ポインタなしにこのタスクを解決する良い方法はありません。

第1.13節GOTO演算子

GOTO演算子は、一般的にアンチパターンとみなされます。[Edgar Dijkstra, *Go To Statement Considered Harmful* (1968)⁸⁷] を参照してください。それにもかかわらず、それは合理的に使用することができます [Donald E. Knuth, *Structured Programming with go to Statements* (1974)⁸⁸] ⁸⁹ を参照してください。

ここには非常に単純な例があります。

```

#include <stdio.h>

int main()
{

```

⁸⁷<http://yurichev.com/mirrors/Dijkstra68.pdf>

⁸⁸<http://yurichev.com/mirrors/KnuthStructuredProgrammingGoTo.pdf>

⁸⁹[Dennis Yurichev, *C/C++ programming language notes*] にもいくつか例があります

```

        printf ("begin\n");
        goto exit;
        printf ("skip me!\n");
exit:
        printf ("end\n");
};

```

MSVC 2012ではは次のようになります。

Listing 1.102: MSVC 2012

```

$SG2934 DB      'begin', 0aH, 00H
$SG2936 DB      'skip me!', 0aH, 00H
$SG2937 DB      'end', 0aH, 00H

_main  PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG2934 ; 'begin'
        call    _printf
        add     esp, 4
        jmp     SHORT $exit$3
        push    OFFSET $SG2936 ; 'skip me!'
        call    _printf
        add     esp, 4
$exit$3:
        push    OFFSET $SG2937 ; 'end'
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main  ENDP

```

`goto` 文は単に `JMP` 命令に置き換えられています。これは同じ効果があります。別の場所への無条件ジャンプです。2番目の `printf()` は、人間の介入、デバッガの使用、またはコードのパッチ適用によってのみ実行できます。

これは簡単なパッチの練習としても役立ちます。Hiewで結果の実行ファイルを開きましょう：

```

Hiew: goto.exe
C:\Polygon\goto.exe  FRO ----- a32 PE .00401000
.00401000: 55          push     ebp
.00401001: 8BEC       mov      ebp, esp
.00401003: 6800304000 push     000403000 ; 'begin' --1
.00401008: FF1590204000 call    printf
.0040100E: 83C404     add      esp, 4
.00401011: EB0E       jz       000401021 --2
.00401013: 6808304000 push     000403008 ; 'skip me!' --3
.00401018: FF1590204000 call    printf
.0040101E: 83C404     add      esp, 4
.00401021: 6814304000 push     000403014 --4
.00401026: FF1590204000 call    printf
.0040102C: 83C404     add      esp, 4
.0040102F: 33C0       xor      eax, eax
.00401031: 5D         pop      ebp
.00401032: C3         ret

```

図 1.32: Hiew

カーソルを JMP (0x410) に設定し、F3 (編集) を押し、0 を 2 回押すと、オペコードが EB 00 になります。

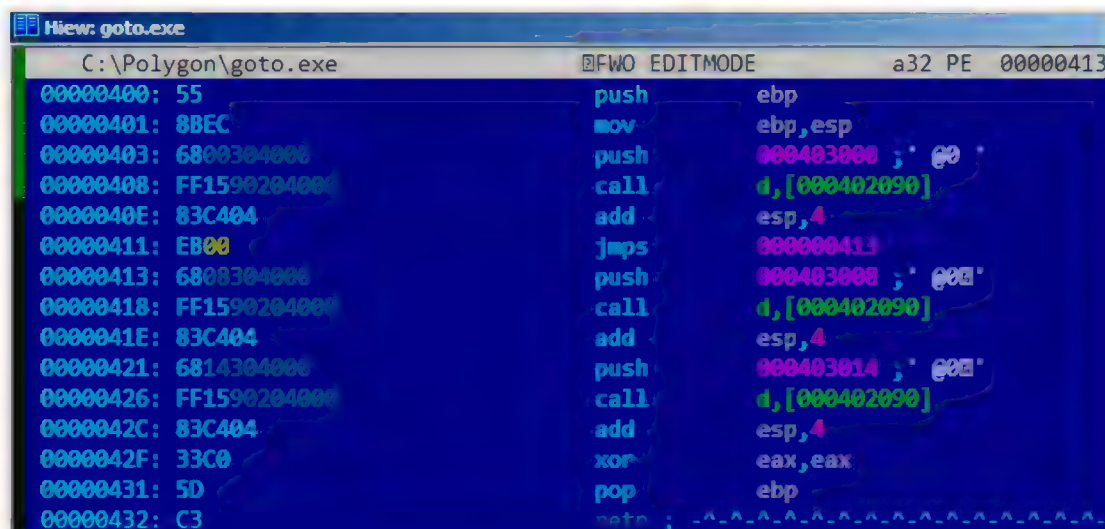


図 1.33: Hiew

JMP オペコードの第2バイトはジャンプの相対オフセットを示し、0は現在の命令の直後のポイントを示します。

したがって、JMP は2番目の printf() 呼び出しをスキップしません。

F9 (保存) を押して終了します。実行ファイルを実行すると、次のように表示されます。

Listing 1.103: Patched executable output

```
C:\...\>goto.exe
begin
skip me!
end
```

JMP 命令を2つの NOP 命令に置き換えることによっても同じ結果が得られます。

NOP のオペコードは 0x90 で、長さは1バイトなので、JMP の代わりに2バイトの命令 (サイズは2バイト) が必要です。

第1.13.1節デッドコード

2番目の printf() 呼び出しは、コンパイラの用語で「デッドコード」とも呼ばれます。

つまり、コードは決して実行されません。したがって、この例を最適化してコンパイルすると、コンパイラは痕跡を残さずに、「デッドコード」を削除します。

Listing 1.104: 最適化 MSVC 2012


```

$SG2981 DB      'begin', 0aH, 00H
$SG2983 DB      'skip me!', 0aH, 00H
$SG2984 DB      'end', 0aH, 00H

_main PROC
    push    OFFSET $SG2981 ; 'begin'
    call    _printf
    push    OFFSET $SG2984 ; 'end'
$exit$4:
    call    _printf
    add     esp, 8
    xor     eax, eax
    ret     0
_main ENDP

```

しかし、コンパイラは「skip me!」文字列を削除するのを忘れていました。

第1.13.2節練習問題

あなたの好きなコンパイラとデバッガを使って同じ結果を達成してみてください。

第1.14節条件付きジャンプ

第1.14.1節シンプルな例

```

#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
}

```

```
    return 0;
};
```

x86

x86 + MSVC

以下は、f_signed() 関数がどうなっているかを示しています。

Listing 1.105: 非最適化 MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    cmp     eax, DWORD PTR _b$[ebp]
    jle     SHORT $LN3@f_signed
    push    OFFSET $SG737          ; 'a>b'
    call    _printf
    add     esp, 4
$LN3@f_signed:
    mov     ecx, DWORD PTR _a$[ebp]
    cmp     ecx, DWORD PTR _b$[ebp]
    jne     SHORT $LN2@f_signed
    push    OFFSET $SG739          ; 'a==b'
    call    _printf
    add     esp, 4
$LN2@f_signed:
    mov     edx, DWORD PTR _a$[ebp]
    cmp     edx, DWORD PTR _b$[ebp]
    jge     SHORT $LN4@f_signed
    push    OFFSET $SG741          ; 'a<b'
    call    _printf
    add     esp, 4
$LN4@f_signed:
    pop     ebp
    ret     0
_f_signed ENDP
```

最初の命令 JLE は、*Jump if Less or Equal* の場合は Jump を表します。言い換えれば、第2オペランドが第1オペランドより大きい場合、制御フローは命令で指定されたアドレスまたはラベルに移ります。第2オペランドが最初のオペランドより小さいためにこの条件がトリガされない場合、制御フローは変更されず、最初の printf() が実行されます。2番目のチェックは、JNE : *Jump if Not Equal* です。オペランドが等しい場合、制御フローは変更されません。

3番目のチェックは、最初のオペランドが2番目のオペランドより大きい場合、または等しい場合は JGE : *Jump if Greater or Equal* です。したがって、3つの条件ジャンプがすべてトリガされた場合、printf() の呼び出しはまったく実行されません。これは特別な

介入なしには不可能です。f_unsigned() 関数を見てみましょう。f_unsigned() 関数は、次のように、JLE および JGE の代わりにJBEおよびJAE命令が使用される点を除いて、f_signed() と同じです。

Listing 1.106: GCC

```

_a$ = 8    ; size = 4
_b$ = 12   ; size = 4
_f_unsigned PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$(ebp)
    cmp     eax, DWORD PTR _b$(ebp)
    jbe     SHORT $LN3@f_unsigned
    push    OFFSET $SG2761    ; 'a>b'
    call    _printf
    add     esp, 4
$LN3@f_unsigned:
    mov     ecx, DWORD PTR _a$(ebp)
    cmp     ecx, DWORD PTR _b$(ebp)
    jne     SHORT $LN2@f_unsigned
    push    OFFSET $SG2763    ; 'a==b'
    call    _printf
    add     esp, 4
$LN2@f_unsigned:
    mov     edx, DWORD PTR _a$(ebp)
    cmp     edx, DWORD PTR _b$(ebp)
    jae     SHORT $LN4@f_unsigned
    push    OFFSET $SG2765    ; 'a<b'
    call    _printf
    add     esp, 4
$LN4@f_unsigned:
    pop     ebp
    ret     0
_f_unsigned ENDP

```

すでに説明したように、分岐命令は異なります。JBE—*Jump if Below or Equal* and JAE—*Jump if Above or Equal* これらの命令（JA/JAE/JB/JBE）は、JG/JGE/JL/JLE とは、符号なしの数字で動作する点が異なります。

JA/JB の代わりに JG/JL が使用されている場合や、その逆の場合は、変数がそれぞれ符号付きか、または符号なしなのかがほぼはっきりします。ここには、もう何も新しくない、main() 関数もあります。

Listing 1.107: main()

```

_main    PROC
    push    ebp
    mov     ebp, esp
    push    2
    push    1
    call    _f_signed
    add     esp, 8
    push    2

```

```
    push    1
    call    _f_unsigned
    add     esp, 8
    xor     eax, eax
    pop     ebp
    ret     0
_main     ENDP
```

x86 + MSVC + OllyDbg

OllyDbg でこの例を実行すると、フラグがどのように設定されているかを見ることができます。符号なしの数値で動作する `f_unsigned()` から始めましょう。

CMP はここで3回実行されますが、同じ引数についてはフラグは毎回同じです。

最初の比較の結果は、

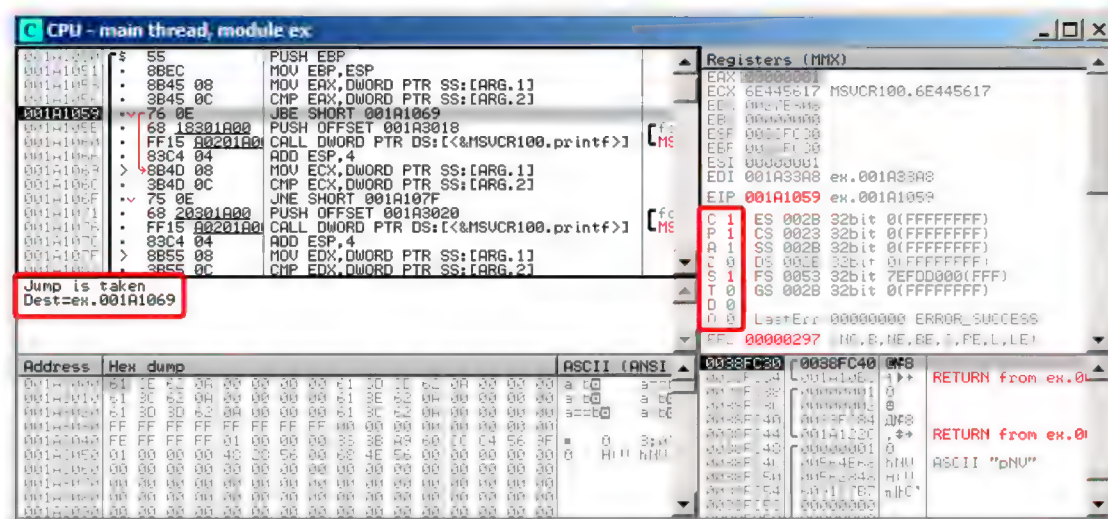


図 1.34: OllyDbg: `f_unsigned()`: 最初の条件付きジャンプ

従って、フラグは、C=1、P=1、A=1、Z=0、S=1、T=0、D=0、O=0です。

これらは OllyDbg では1文字の略号で命名されています。

OllyDbg は、(JBE) ジャンプがトリガーされることを示唆しています。実際に、インテルのマニュアル (8.1.4 on page 564) を調べると、CF=1またはZF=1の場合、JBEが起動することがわかります。条件はここに当てはまるので、ジャンプが開始されます。

次の条件付きジャンプは、

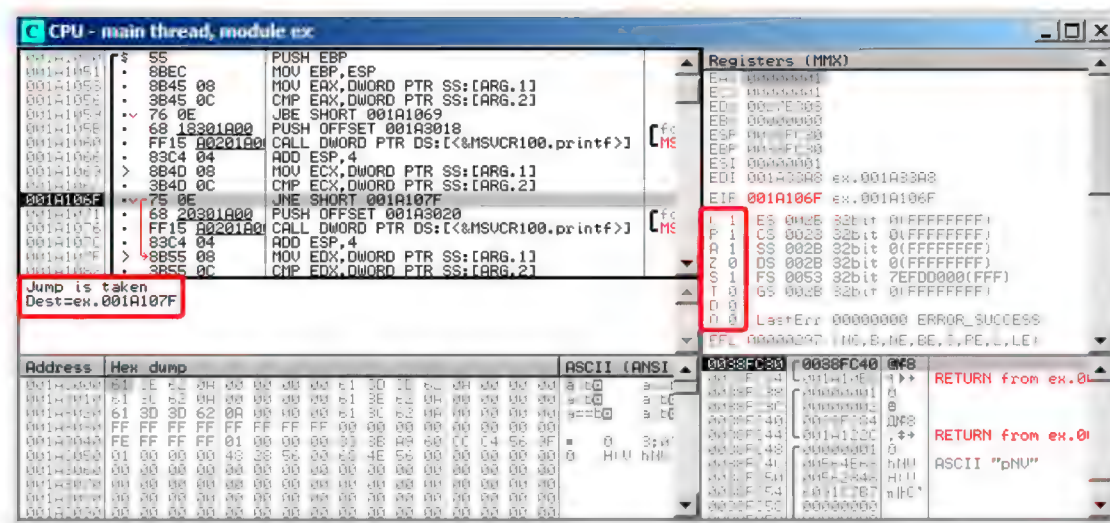


図 1.35: OllyDbg: f_unsigned(): 2番目の条件付きジャンプ

OllyDbg は、JNZ がトリガーされることを示唆しています。実際、ZF=0（ゼロフラグ）の場合、JNZが起動します。

3番目の条件付きジャンプは、JNB です。

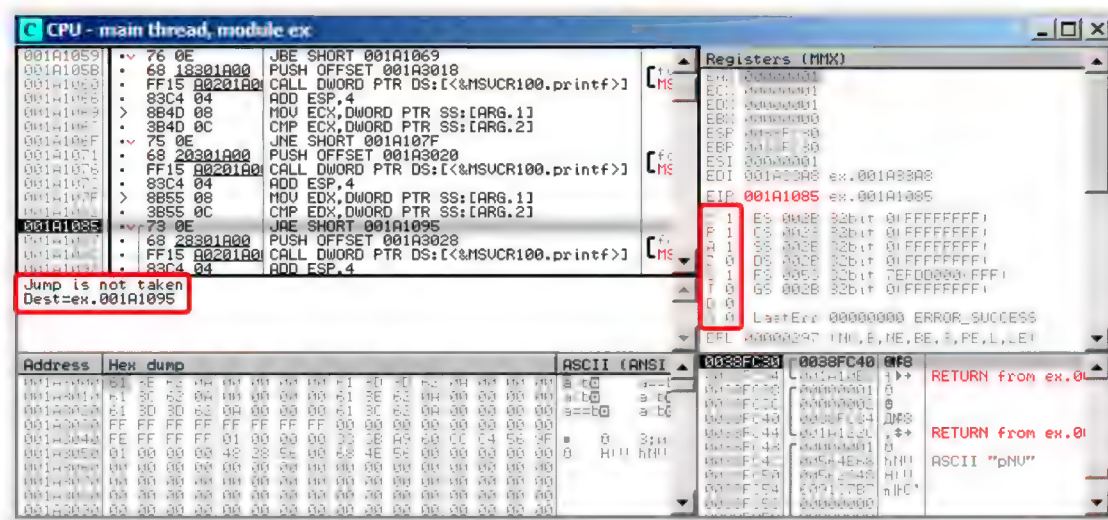


図 1.36: OllyDbg: f_unsigned(): 3番目の条件付きジャンプ

インテルのマニュアル (8.1.4 on page 564) では、CF=0 (キャリーフラグ) の場合に JNB が起動することがわかります。今回は当てはまらないので、3番目の printf() が実行されます。

The screenshot displays the CPU window for the main thread in module 'ex'. The assembly pane shows the following instructions:

```

001A1009 55      PUSH EBP
001A100A 8BEC    MOV EBP, ESP
001A100B 8B45 08  MOV EAX, DWORD PTR SS:[ARG.1]
001A100C 8B45 0C  MOV EAX, DWORD PTR SS:[ARG.2]
001A100D 7E 06   JLE SHORT 001A1019
001A100E 68 00301A00 PUSH OFFSET 001A3000
001A100F FF15 00201A00 CALL DWORD PTR DS:[<MSUCR100.printf>]
001A1010 83C4 04  ADD ESP, 4
001A1011 8B4D 08  MOV ECX, DWORD PTR SS:[ARG.1]
001A1012 8B4D 0C  MOV ECX, DWORD PTR SS:[ARG.2]
001A1013 7E 06   JLE SHORT 001A102F
001A1014 68 00301A00 PUSH OFFSET 001A3000
001A1015 FF15 00201A00 CALL DWORD PTR DS:[<MSUCR100.printf>]
001A1016 83C4 04  ADD ESP, 4
001A1017 MOV EDI, DWORD PTR SS:[ARG.1]
001A1018 MOV EDI, DWORD PTR SS:[ARG.2]
001A1019 8B45 0C  MOV EAX, DWORD PTR SS:[ARG.2]

```

The registers pane shows the following values:

Register	Value	Comment
EAX	00000000	
ECX	00000000	
EDX	00000000	
ESI	00000000	
EDI	001A0909	ex.001A0909
EIP	001A1009	ex.001A1009

The status bar indicates: "Jump is taken" and "Dest=ex.001A1019".

インテルマニュアル (8.1.4 on page 564) では、ZF=1または SF≠OF の場合にこの命令が起動されることがわかりました。私たちの場合では SF≠OF が、ジャンプが起動されます。

2番目の JNZ 条件付きジャンプはZF=0の場合（ゼロ・フラグ）に起動します。

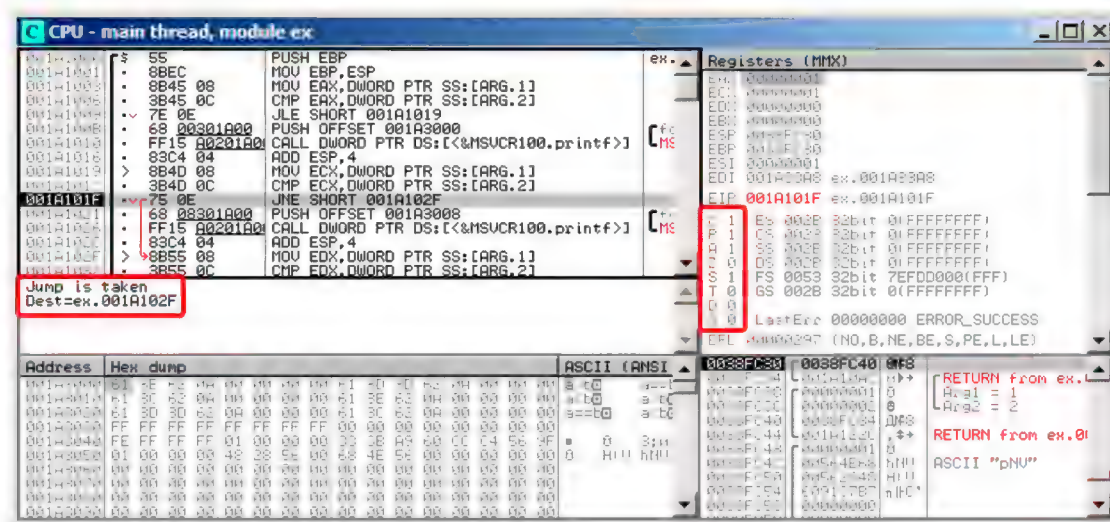


図 1.38: OllyDbg: f_signed(): 2番目の条件付きジャンプ

第3の条件付きジャンプ JGE は、SF=OFの場合にのみ実行されるため、起動しません。今回は、当てはまりません。

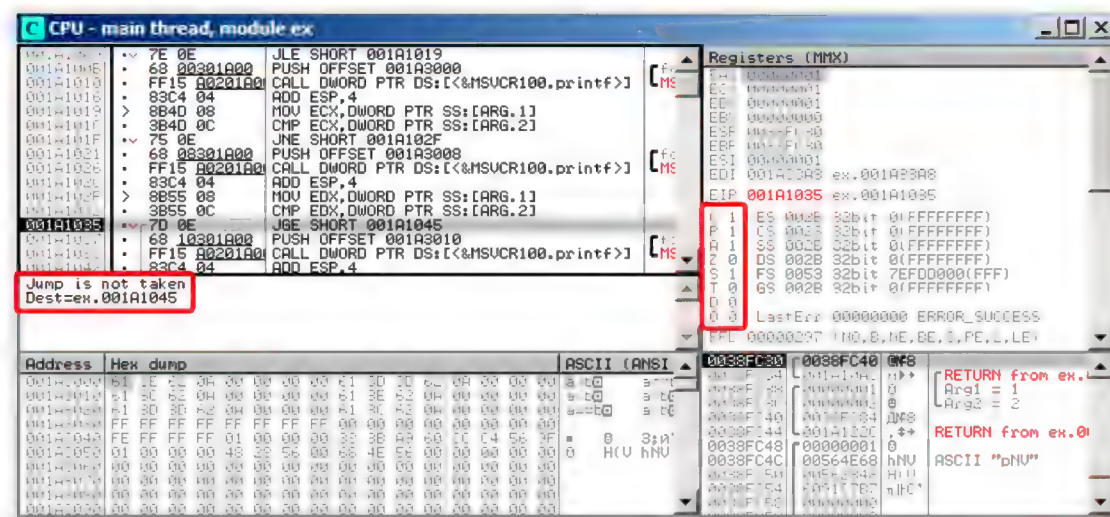


図 1.39: OllyDbg: f_signed(): 3番目の条件付きジャンプ

x86 + MSVC + Hiew

入力値にかかわらず、f_unsigned() 関数が常に「a==b」を出力するように、実行可能ファイルにパッチを当てることができます。ここで、Hiewでどのように見えるか見てみましょう。

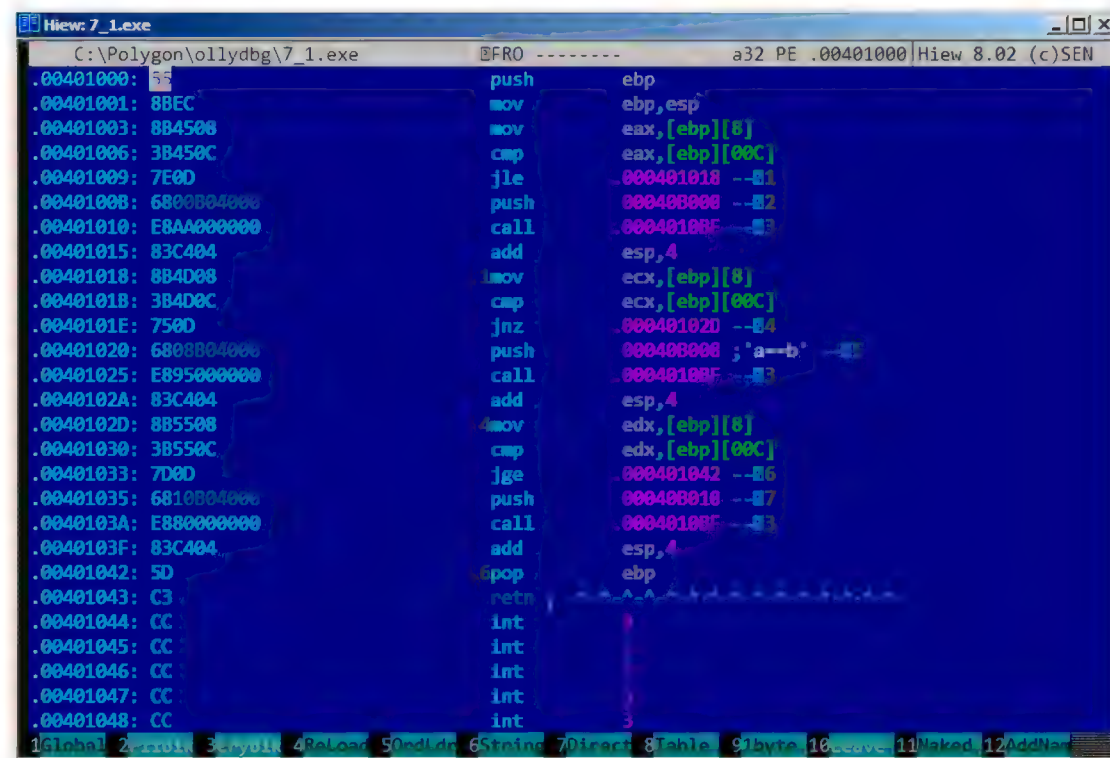


図 1.40: Hiew: f_unsigned() 関数

本質的には、次の3つのタスクを実行する必要があります。

- 最初のジャンプが常に起動しなければならない
- 2番目のジャンプが決して起動してはならない
- 3番目のジャンプが常にト起動しなければならない

したがって、コードフローは常に2番目の printf() を通過し、「a==b」を出力するように指示できます。

3つの命令（またはバイト）をパッチする必要があります。

- 最初のジャンプはJMPになりますが、jump offsetは同じままです。
- 2回目のジャンプがトリガされることもありますが、いずれにしても次の命令にジャンプします。

なぜなら、`jump offset`を0に設定しているからです。これらの命令では、ジャンプオフセットが次の命令のアドレスに追加されます。オフセットが0の場合、ジャンプは制御を次の命令に移します。

- 私たちが最初のもと同様に JMP を置き換える3番目のジャンプは、常に起動します。

変更されたコードは次のとおりです。

```

Hiew: 7_1.exe
C:\Polygon\ollydbg\7_1.exe  FWO EDITMODE  a32 PE  00000434 Hiew 8.02 (c)SEN
00000400: 55          push     ebp
00000401: 8BEC       mov     ebp, esp
00000403: 8B4506     mov     eax, [ebp+8]
00000406: 3B450C     cmp     eax, [ebp+00C]
00000409: EB0D      jmps     00000418
0000040B: 68000400  push     00040000
00000410: E8AA0000  call    0000040F
00000415: 83C404     add     esp, 4
00000418: 8B4D08     mov     ecx, [ebp+8]
0000041B: 3B4D0C     cmp     ecx, [ebp+00C]
0000041E: 7508      jnz     00000420
00000420: 68000400  push     00040000
00000425: E8950000  call    0000041F
0000042A: 83C404     add     esp, 4
0000042D: 8B5508     mov     edx, [ebp+8]
00000430: 3B550C     cmp     edx, [ebp+00C]
00000433: EB0D      jmps     00000442
00000435: 68100400  push     00040010
0000043A: E8800000  call    0000041F
0000043F: 83C404     add     esp, 4
00000442: 5D        pop     ebp
00000443: C3        retn
00000444: CC        int3
00000445: CC        int3
00000446: CC        int3
00000447: CC        int3
00000448: CC        int3

```

図 1.41: Hiew: let's modify the f_unsigned() function

これらのジャンプのいずれかを変更することができなければ、printf() 呼び出しを1回だけ実行したいのですが、何回か実行することになるでしょう。

非最適化 GCC

非最適化 GCC 4.4.1 はほとんど同じコードを生成しますが、printf() ではなく puts() (1.5.3 on page 27) が生成されます。

最適化 GCC

実行される度にフラグが同じ値を持つ場合、鋭い読者はなぜ CMP が何度も実行されるのかと尋ねるかもしれません。

おそらく、最適化されたMSVCではこうはできませんが、GCC 4.8.1の最適化はより深刻です。

Listing 1.108: GCC 4.8.1 f_signed()


```

f_signed:
    mov     eax, DWORD PTR [esp+8]
    cmp     DWORD PTR [esp+4], eax
    jg      .L6
    je      .L7
    jge     .L1
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
    jmp     puts
.L6:
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
    jmp     puts
.L1:
    rep ret
.L7:
    mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
    jmp     puts

```

また、CALL puts / RETN の代わりにここに JMPを入れています。

この種のトリックは後で説明します：[1.15.1 on page 190](#)

この種のx86コードは、まれです。MSVC 2012のように、そのようなコードを生成することはできません。一方、アセンブリ言語プログラマは、Jcc 命令を積み重ねることができるという事実を十分に認識しています。

だから、どこかでそのような積み重ねを見ると、コードは手書きの可能性が高いです。

f_unsigned() 関数は巧妙に短いものではありません：

Listing 1.109: GCC 4.8.1 f_unsigned()

```

f_unsigned:
    push    esi
    push    ebx
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    mov     ebx, DWORD PTR [esp+36]
    cmp     esi, ebx
    ja      .L13
    cmp     esi, ebx ; この命令は削除することができます
    je      .L14
.L10:
    jb      .L15
    add     esp, 20
    pop     ebx
    pop     esi
    ret
.L15:
    mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
    add     esp, 20
    pop     ebx
    pop     esi
    jmp     puts
.L13:
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"

```



```

        call    puts
        cmp     esi, ebx
        jne     .L10
.L14:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts

```

それにもかかわらず、3つではなく2つの CMP 命令があります。

したがって、GCC 4.8.1の最適化アルゴリズムはまだ完璧ではないでしょう。

ARM

32-bit ARM

最適化 Keil 6/2013 (ARMモード)

Listing 1.110: 最適化 Keil 6/2013 (ARMモード)

```

.text:000000B8                                EXPORT f_signed
.text:000000B8                                f_signed                ; CODE XREF: main+C
.text:000000B8 70 40 2D E9                    STMFD    SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1                    MOV      R4, R1
.text:000000C0 04 00 50 E1                    CMP      R0, R4
.text:000000C4 00 50 A0 E1                    MOV      R5, R0
.text:000000C8 1A 0E 8F C2                    ADRGT    R0, aAB                ; "a>b\n"
.text:000000CC A1 18 00 CB                    BLGT     __2printf
.text:000000D0 04 00 55 E1                    CMP      R5, R4
.text:000000D4 67 0F 8F 02                    ADREQ    R0, aAB_0             ; "a==b\n"
.text:000000D8 9E 18 00 0B                    BLEQ     __2printf
.text:000000DC 04 00 55 E1                    CMP      R5, R4
.text:000000E0 70 80 BD A8                    LDMGEFD  SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8                    LDMFD    SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2                    ADR      R0, aAB_1             ; "a<b\n"
.text:000000EC 99 18 00 EA                    B        __2printf
.text:000000EC                                ; End of function f_signed

```

ARMモードの多くの命令は、特定のフラグがセットされている場合にのみ実行できます。例えば、これは数字を比較するときによく使用されます。

例えば、ADD 命令は実際には内部で ADDAL と名付けられ、ALは常に、すなわち常に実行する。述語は、32ビットARM命令の4つの上位ビット（条件フィールド）でエンコードされます。無条件ジャンプの B 命令は、実際には条件付きで他の条件ジャンプと同様にエンコードされますが、条件フィールドには AL があり、フラグを無視して常に実行することを意味します。

ADRG 命令は ADR と同じように動作しますが、前の CMP 命令が2つ（大きい方）を比較しながら、他の命令より大きな数値の1つを検出した場合にのみ実行されます。

次の BLGT 命令は BL と同じように動作し、比較の結果が（より大きい）場合にのみ実行されます。ADRG 命令は文字列 `a>b\n` へのポインタを R0 に書き込み、BLGT は `printf()` を呼び出します。したがって、-GT の後に続く命令は、R0 (*a*) の値が R4 (*b*) の値より大きい場合にのみ実行されます。

ADREQ 命令と BLEQ 命令が順方向に進みます。それらは ADR と BL のように動作しますが、最後の比較時にオペランドが等しい場合にのみ実行されます。`printf()` の実行によってフラグが改ざんされた可能性があるため、別の CMP がその前に配置されます。

次に、LDMGEFD を参照してください。この命令は LDMFD⁹⁰ のように機能しますが、一方の値が他方の値より大きい場合等しい場合にのみ実行されます。LDMGEFD SP!, {R4-R6,PC} 命令は関数エピローグのように動作しますが、 $a \geq b$ の場合にのみトリガされ、その後に関数の実行が終了します。

しかし、その条件が満たされない場合、すなわち $a < b$ の場合、制御フローは次の「LDMFD SP!, {R4-R6,LR}」命令に続き、これはもう1つの関数エピローグです。この命令は、R4-R6 だけでなく **PC!** の代わりに **LR** も登録されているため、関数からは戻りません。最後の2つの命令は、文字列 `«a<b\n»` を唯一の引数として `printf()` を呼び出します。`printf()` セクション (1.8.2 on page 68) の関数の戻り値ではなく、`printf()` 関数への無条件ジャンプを調べました。

`f_unsigned` は類似しており、ADRI, BLHI、および LDMCSFD 命令のみが使用されています。これらの述部 (*HI* = *Unsigned higher*, *CS* = *Carry Set (greater than or equal)*) は、前に説明したものと類似しています。

`main()` 関数にはそんなに新しい点はありません。

Listing 1.111: `main()`

```
.text:00000128      EXPORT main
.text:00000128      main
.text:00000128 10 40 2D E9      STMFD    SP!, {R4,LR}
.text:0000012C 02 10 A0 E3      MOV     R1, #2
.text:00000130 01 00 A0 E3      MOV     R0, #1
.text:00000134 DF FF FF EB      BL      f_signed
.text:00000138 02 10 A0 E3      MOV     R1, #2
.text:0000013C 01 00 A0 E3      MOV     R0, #1
.text:00000140 EA FF FF EB      BL      f_unsigned
.text:00000144 00 00 A0 E3      MOV     R0, #0
.text:00000148 10 80 BD E8      LDMFD    SP!, {R4,PC}
.text:00000148      ; End of function main
```

これは、ARMモードでの条件付きジャンプを取り除く方法です。

なぜこれがよいのでしょうか？以下を読んでください：?? on page ??

x86では、CMOVcc 命令以外は MOV と同じですが、通常は CMP によって設定された特定のフラグが設定されている場合にのみ実行されます。

最適化 Keil 6/2013 (Thumbモード)

⁹⁰LDMFD

Listing 1.112: 最適化 Keil 6/2013 (Thumbモード)

```

.text:00000072      f_signed ; CODE XREF: main+6
.text:00000072 70 B5      PUSH    {R4-R6,LR}
.text:00000074 0C 00      MOVVS   R4, R1
.text:00000076 05 00      MOVVS   R5, R0
.text:00000078 A0 42      CMP     R0, R4
.text:0000007A 02 DD      BLE     loc_82
.text:0000007C A4 A0      ADR     R0, aAB          ; "a>b\n"
.text:0000007E 06 F0 B7 F8    BL      __2printf
.text:00000082
.text:00000082      loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42      CMP     R5, R4
.text:00000084 02 D1      BNE     loc_8C
.text:00000086 A4 A0      ADR     R0, aAB_0        ; "a==b\n"
.text:00000088 06 F0 B2 F8    BL      __2printf
.text:0000008C
.text:0000008C      loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42      CMP     R5, R4
.text:0000008E 02 DA      BGE     locret_96
.text:00000090 A3 A0      ADR     R0, aAB_1        ; "a<b\n"
.text:00000092 06 F0 AD F8    BL      __2printf
.text:00000096
.text:00000096      locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD      POP     {R4-R6,PC}
.text:00000096      ; End of function f_signed

```

Thumbモードの B 命令だけが条件コードで補完されるため、Thumbコードはより一般的に見えます。

BLE は通常の条件ジャンプであり、*Less than or Equal* の意味です。BNE は *Not Equal* の意味です。BGE は *Greater than or Equal* の意味です。

f_unsigned は似ていますが、符号なしの値を扱う際には、BLS (*Unsigned lower or same*) および BCS (*Carry Set (Greater than or equal)*) 命令しか使用されません。

ARM64: 最適化 GCC (Linaro) 4.9

Listing 1.113: f_signed()

```

f_signed:
; w0=a, w1=b
    cmp     w0, w1
    bgt     .L19      ; 大きければ (a>b) 分岐
    beq     .L20      ; 等しければ (a==b) 分岐
    bge     .L15      ; 大きい、または等しければ分岐 (a>=b) (不可能)
; a<b
    adrp    x0, .LC11      ; "a<b"
    add     x0, x0, :lo12:LC11
    b       puts
.L19:
    adrp    x0, .LC9       ; "a>b"
    add     x0, x0, :lo12:LC9

```

```

b      puts
.L15:  ; ここに来るのは不可能
      ret
.L20:
      adrp    x0, .LC10      ; "a==b"
      add     x0, x0, :lo12:LC10
      b      puts

```

Listing 1.114: f_unsigned()

```

f_unsigned:
      stp     x29, x30, [sp, -48]!
; W0=a, W1=b
      cmp     w0, w1
      add     x29, sp, 0
      str     x19, [sp,16]
      mov     w19, w0
      bhi     .L25      ; 大きければ (a>b) 分岐
      cmp     w19, w1
      beq     .L26      ; 等しければ (a==b) 分岐
.L23:
      bcc     .L27      ; キャリーフラグがクリアされてたら分岐 (小さければ) (a<b)
; 関数エピローグ、ここに来るのは不可能
      ldr     x19, [sp,16]
      ldp     x29, x30, [sp], 48
      ret
.L27:
      ldr     x19, [sp,16]
      adrp    x0, .LC11      ; "a<b"
      ldp     x29, x30, [sp], 48
      add     x0, x0, :lo12:LC11
      b      puts
.L25:
      adrp    x0, .LC9      ; "a>b"
      str     x1, [x29,40]
      add     x0, x0, :lo12:LC9
      bl      puts
      ldr     x1, [x29,40]
      cmp     w19, w1
      bne     .L23      ; 等しくなければ分岐
.L26:
      ldr     x19, [sp,16]
      adrp    x0, .LC10      ; "a==b"
      ldp     x29, x30, [sp], 48
      add     x0, x0, :lo12:LC10
      b      puts

```

コメントはこの本の著者によって追加されました。目立ったことは、コンパイラはいくつかの条件がまったく不可能であることを認識していないため、決して実行できない場所ではデッドコードがあることです。

練習問題

これらの機能をサイズが少なくなるように手動で最適化し、新しい命令を追加せずに冗長な命令を削除してください。

MIPS

1つの特徴的なMIPS機能は、フラグが存在しないことです。明らかに、データ依存性の分析を簡素化するために行われました。

x86には SETcc に似た命令があります。SLT (「Set on Less Than」: 符号付きバージョン) と SLTU (符号なしバージョン) です。これらの命令は、条件が真であれば宛先レジスタの値を1に設定し、そうでない場合は0に設定します。

宛先レジスタは、BEQ (「Branch on Equal」) または BNE (「Branch on Not Equal」) を使用してチェックされ、ジャンプが発生することがあります。したがって、この命令ペアは比較および分岐のためにMIPSで使用されなければなりません。最初に関数の符号付きバージョンから始めましょう。

Listing 1.115: 非最適化 GCC 4.4.5 (IDA)

```
.text:00000000 f_signed: # CODE XREF: main+18
.text:00000000
.text:00000000 var_10 = -0x10
.text:00000000 var_8 = -8
.text:00000000 var_4 = -4
.text:00000000 arg_0 = 0
.text:00000000 arg_4 = 4
.text:00000000
.text:00000000      addiu   $sp, -0x20
.text:00000004      sw     $ra, 0x20+var_4($sp)
.text:00000008      sw     $fp, 0x20+var_8($sp)
.text:0000000C      move   $fp, $sp
.text:00000010      la     $gp, __gnu_local_gp
.text:00000018      sw     $gp, 0x20+var_10($sp)
; 入力値をローカルスタックに格納する
.text:0000001C      sw     $a0, 0x20+arg_0($fp)
.text:00000020      sw     $a1, 0x20+arg_4($fp)
; リロードする
.text:00000024      lw     $v1, 0x20+arg_0($fp)
.text:00000028      lw     $v0, 0x20+arg_4($fp)
; $v0=b
; $v1=a
.text:0000002C      or     $at, $zero ; NOP
; これは疑似命令です。実際は、"slt $v0,$v0,$v1" です。
; $v0<$v1 (b<a) なら $v0に1が設定され、そうでなければ0が設定されます
.text:00000030      slt    $v0, $v1
; 条件が真でない場合、loc_5cにジャンプします。
; これは疑似命令です。実際は、"beq $v0,$zero,loc_5c" です。
.text:00000034      beqz   $v0, loc_5C
; "a>b" を表示して終了します
.text:00000038      or     $at, $zero ; 分岐遅延スロット、NOP
.text:0000003C      lui    $v0, (unk_230 >> 16) # "a>b"
```

```

.text:00000040      addiu    $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044      lw        $v0, (puts & 0xFFFF)($gp)
.text:00000048      or        $at, $zero ; NOP
.text:0000004C      move     $t9, $v0
.text:00000050      jalr     $t9
.text:00000054      or        $at, $zero ; 分岐遅延スロット、NOP
.text:00000058      lw        $gp, 0x20+var_10($fp)
.text:0000005C      loc_5C:                                     # CODE XREF: f_signed+34
.text:0000005C      lw        $v1, 0x20+arg_0($fp)
.text:00000060      lw        $v0, 0x20+arg_4($fp)
.text:00000064      or        $at, $zero ; NOP
; a==bであるかどうかを調べ、真でなければloc_90にジャンプします。
.text:00000068      bne      $v1, $v0, loc_90
.text:0000006C      or        $at, $zero ; 分岐遅延スロット、NOP
; 条件が真なので、"a==b" をプリントして終了する
.text:00000070      lui      $v0, (aAB >> 16) # "a==b"
.text:00000074      addiu    $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000078      lw        $v0, (puts & 0xFFFF)($gp)
.text:0000007C      or        $at, $zero ; NOP
.text:00000080      move     $t9, $v0
.text:00000084      jalr     $t9
.text:00000088      or        $at, $zero ; 分岐遅延スロット、NOP
.text:0000008C      lw        $gp, 0x20+var_10($fp)
.text:00000090      loc_90:                                     # CODE XREF: f_signed+68
.text:00000090      lw        $v1, 0x20+arg_0($fp)
.text:00000094      lw        $v0, 0x20+arg_4($fp)
.text:00000098      or        $at, $zero ; NOP
; $v1<$v0 (a <b) かどうかをチェックし、条件が真であれば $v0を1に設定する
.text:0000009C      slt      $v0, $v1, $v0
; 条件が真でない場合 (すなわち、$v0==0)、loc_c8にジャンプします
.text:000000A0      beqz     $v0, loc_c8
.text:000000A4      or        $at, $zero ; 分岐遅延スロット、NOP
; 条件が真であれば、"a<b" をプリントして終了します
.text:000000A8      lui      $v0, (aAB_0 >> 16) # "a<b"
.text:000000AC      addiu    $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:000000B0      lw        $v0, (puts & 0xFFFF)($gp)
.text:000000B4      or        $at, $zero ; NOP
.text:000000B8      move     $t9, $v0
.text:000000BC      jalr     $t9
.text:000000C0      or        $at, $zero ; 分岐遅延スロット、NOP
.text:000000C4      lw        $gp, 0x20+var_10($fp)
.text:000000C8      ; 3つの条件はすべて偽でした。
.text:000000C8      loc_c8:                                     # CODE XREF:
                                          f_signed+A0
.text:000000C8      move     $sp, $fp
.text:000000CC      lw        $ra, 0x20+var_4($sp)
.text:000000D0      lw        $fp, 0x20+var_8($sp)
.text:000000D4      addiu    $sp, 0x20
.text:000000D8      jr       $ra
.text:000000DC      or        $at, $zero ; 分岐遅延スロット、NOP

```

```
.text:000000DC # End of function f_signed
```

SLT REG0, REG0, REG1 は、IDAによって短縮形式 SLT REG0, REG1 に縮小されます。実際には BEQ REG, \$ZERO, LABEL の BEQZ 擬似命令もあります（「Branch if Equal to Zero」）。

符号なしバージョンはまったく同じですが、SLT の代わりに SLTU（符号なしバージョン、したがって「U」という名前）が使用されます。

Listing 1.116: 非最適化 GCC 4.4.5 (IDA)

```
.text:000000E0 f_unsigned: # CODE XREF: main+28
.text:000000E0
.text:000000E0 var_10 = -0x10
.text:000000E0 var_8 = -8
.text:000000E0 var_4 = -4
.text:000000E0 arg_0 = 0
.text:000000E0 arg_4 = 4
.text:000000E0
.text:000000E0      addiu    $sp, -0x20
.text:000000E4      sw      $ra, 0x20+var_4($sp)
.text:000000E8      sw      $fp, 0x20+var_8($sp)
.text:000000EC      move    $fp, $sp
.text:000000F0      la      $gp, __gnu_local_gp
.text:000000F8      sw      $gp, 0x20+var_10($sp)
.text:000000FC      sw      $a0, 0x20+arg_0($fp)
.text:00000100      sw      $a1, 0x20+arg_4($fp)
.text:00000104      lw      $v1, 0x20+arg_0($fp)
.text:00000108      lw      $v0, 0x20+arg_4($fp)
.text:0000010C      or      $at, $zero
.text:00000110      sltu    $v0, $v1
.text:00000114      beqz    $v0, loc_13C
.text:00000118      or      $at, $zero
.text:0000011C      lui     $v0, (unk_230 >> 16)
.text:00000120      addiu    $a0, $v0, (unk_230 & 0xFFFF)
.text:00000124      lw      $v0, (puts & 0xFFFF)($gp)
.text:00000128      or      $at, $zero
.text:0000012C      move    $t9, $v0
.text:00000130      jalr    $t9
.text:00000134      or      $at, $zero
.text:00000138      lw      $gp, 0x20+var_10($fp)
.text:0000013C
.text:0000013C loc_13C: # CODE XREF: f_unsigned+34
.text:0000013C      lw      $v1, 0x20+arg_0($fp)
.text:00000140      lw      $v0, 0x20+arg_4($fp)
.text:00000144      or      $at, $zero
.text:00000148      bne     $v1, $v0, loc_170
.text:0000014C      or      $at, $zero
.text:00000150      lui     $v0, (aAB >> 16) # "a==b"
.text:00000154      addiu    $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000158      lw      $v0, (puts & 0xFFFF)($gp)
.text:0000015C      or      $at, $zero
.text:00000160      move    $t9, $v0
```



```

.text:00000164      jalr      $t9
.text:00000168      or        $at, $zero
.text:0000016C      lw        $gp, 0x20+var_10($fp)
.text:00000170
.text:00000170 loc_170:      # CODE XREF: f_unsigned+68
.text:00000170      lw        $v1, 0x20+arg_0($fp)
.text:00000174      lw        $v0, 0x20+arg_4($fp)
.text:00000178      or        $at, $zero
.text:0000017C      sltu      $v0, $v1, $v0
.text:00000180      beqz      $v0, loc_1A8
.text:00000184      or        $at, $zero
.text:00000188      lui      $v0, (aAB_0 >> 16) # "a<b"
.text:0000018C      addiu     $a0, $v0, (aAB_0 & 0xFFFF) # "a<b"
.text:00000190      lw        $v0, (puts & 0xFFFF)($gp)
.text:00000194      or        $at, $zero
.text:00000198      move     $t9, $v0
.text:0000019C      jalr      $t9
.text:000001A0      or        $at, $zero
.text:000001A4      lw        $gp, 0x20+var_10($fp)
.text:000001A8
.text:000001A8 loc_1A8:      # CODE XREF: f_unsigned+A0
.text:000001A8      move     $sp, $fp
.text:000001AC      lw        $ra, 0x20+var_4($sp)
.text:000001B0      lw        $fp, 0x20+var_8($sp)
.text:000001B4      addiu     $sp, 0x20
.text:000001B8      jr        $ra
.text:000001BC      or        $at, $zero
.text:000001BC # End of function f_unsigned

```

第1.14.2節絶対値の計算

簡単な関数の例。

```

int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
};

```

最適化 MSVC

これは普通、どのようにコードが生成されるのかを示したものです。

Listing 1.117: 最適化 MSVC 2012 x64

```

i$ = 8
my_abs PROC
; ECX = input
    test     ecx, ecx
; 入力値の符号をチェックする

```

```

; 符号が正の場合はNEG命令をスキップする
    jns     SHORT $LN2@my_abs
; 値を反転する
    neg     ecx
$LN2@my_abs:
; EAXに結果を準備
    mov     eax, ecx
    ret     0
my_abs    ENDP

```

GCC 4.9はほとんど同じです。

最適化 Keil 6/2013: Thumbモード

Listing 1.118: 最適化 Keil 6/2013: Thumbモード

```

my_abs PROC
    CMP     r0,#0
; 入力値はゼロに等しいかゼロより大きいのか
; RSBS命令をスキップする
    BGE     |L0.6|
; 入力値を0から減算する
    RSBS    r0,r0,#0
|L0.6|
    BX      lr
    ENDP

```

ARMにはネゲート命令がないため、Keilコンパイラは「逆引き命令」を使用します。これは逆のオペランドで減算するだけです。

最適化 Keil 6/2013: ARMモード

ARMモードでは、いくつかの命令に条件コードを追加することができます。そのため、Keilコンパイラは次のように処理します。

Listing 1.119: 最適化 Keil 6/2013: ARMモード

```

my_abs PROC
    CMP     r0,#0
; 入力値が0より小さい場合にのみ"逆引き"命令を実行する
    RSBLT   r0,r0,#0
    BX      lr
    ENDP

```

今回は条件付きジャンプはありません。これは良いですね。: ?? on page ??

非最適化 GCC 4.9 (ARM64)

ARM64には、否定するための命令 NEG があります。

Listing 1.120: 最適化 GCC 4.9 (ARM64)

```

my_abs:
    sub    sp, sp, #16
    str    w0, [sp,12]
    ldr    w0, [sp,12]
; 入力値をWZRレジスタの内容と比較する
; (常にゼロを保持する)
    cmp    w0, wzr
    bge    .L2
    ldr    w0, [sp,12]
    neg    w0, w0
    b      .L3
.L2:
    ldr    w0, [sp,12]
.L3:
    add    sp, sp, 16
    ret

```

MIPS

Listing 1.121: 最適化 GCC 4.4.5 (IDA)

```

my_abs:
; $a0<0ならジャンプ:
    bltz    $a0, locret_10
; 入力値 ($a0) を $v0に設定してリターン
    move    $v0, $a0
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP
locret_10:
; 入力値を反転し、$v0に保存する:
    jr      $ra
; これは疑似命令です。実際には、"subu $v0,$zero,$a0" ($v0=0-$a0) です。
    negu    $v0, $a0

```

ここでは BLTZ (「Branch if Less Than Zero」) という新しい命令があります。

NEGU 疑似命令もあります。これはゼロからの減算だけです。SUBU と NEGU の両方の「U」接尾辞は、整数オーバーフローの場合に発生する例外がないことを意味します。

Branchless version?

このコードを分岐がないバージョンにすることもできます。これについては、後述の ?? on page ?? を参照してください。

第1.14.3節三項条件演算子

C/C++ の三項条件演算子の構文は次のとおりです。

```
expression ? expression : expression
```

次に例を示します。

```
const char* f (int a)
{
    return a==10 ? "it is ten" : "it is not ten";
};
```

x86

古いコンパイラと最適化していないコンパイラは、if/else 文が使用されたかのようにアセンブリコードを生成します。

Listing 1.122: 非最適化 MSVC 2008

```
$SG746 DB      'it is ten', 00H
$SG747 DB      'it is not ten', 00H

tv65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f      PROC
    push    ebp
    mov     ebp, esp
    push    ecx
; 入力値と10を比較
    cmp     DWORD PTR _a$[ebp], 10
; 同じでなければ、$LN3@fにジャンプ
    jne     SHORT $LN3@f
; 文字列へのポインタを一時変数に保存
    mov     DWORD PTR tv65[ebp], OFFSET $SG746 ; 'it is ten'
; exitにジャンプ
    jmp     SHORT $LN4@f
$LN3@f:
; 文字列へのポインタを一時変数に保存
    mov     DWORD PTR tv65[ebp], OFFSET $SG747 ; 'it is not ten'
$LN4@f:
; exitです。文字列へのポインタを一時変数からEAXにコピー
    mov     eax, DWORD PTR tv65[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f      ENDP
```

Listing 1.123: 最適化 MSVC 2008

```
$SG792 DB      'it is ten', 00H
$SG793 DB      'it is not ten', 00H

_a$ = 8 ; size = 4
_f      PROC
; 入力値と10を比較
    cmp     DWORD PTR _a$[esp-4], 10
    mov     eax, OFFSET $SG792 ; 'it is ten'
; 同じなら $LN4@fにジャンプ
```

```

        je      SHORT $LN4@f
        mov     eax, OFFSET $SG793 ; 'it is not ten'
$LN4@f:
        ret     0
_f      ENDP

```

新しいコンパイラはより簡潔です。

Listing 1.124: 最適化 MSVC 2012 x64

```

$SG1355 DB      'it is ten', 00H
$SG1356 DB      'it is not ten', 00H

a$ = 8
f      PROC
; 両方の文字列のポインタをロードする
        lea     rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
        lea     rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; 入力値と10を比較
        cmp     ecx, 10
; 同じなら、値をRDXからコピー ("it is ten")
; 異なるなら、何もしない。文字列へのポインタ
; "it is not ten" はまだRAXにある。
        cmove   rax, rdx
        ret     0
f      ENDP

```

x86用の最適化 GCC 4.8も CMOVcc 命令を使用し、非最適化GCC 4.8は条件付きジャンプを使用します。

ARM

ARMモード用の 最適化 Keilでは、条件付き命令 ADRcc を使います。

Listing 1.125: 最適化 Keil 6/2013 (ARMモード)

```

f PROC
; 入力値と10を比較
        CMP     r0,#0xa
; 結果が同じか比較し同じなら、"it is ten" 文字列へのポインタをR0にコピー
        ADREQ   r0,|L0.16| ; "it is ten"
; 結果が同じか比較し異なるなら、"it is not ten" 文字列へのポインタをR0にコピー
        ADRNE   r0,|L0.28| ; "it is not ten"
        BX      lr
        ENDP

|L0.16|
        DCB     "it is ten",0
|L0.28|
        DCB     "it is not ten",0

```

手動で介入しなければ、2つの命令 ADREQ と ADRNE を同じときに実行することはできません。

Thumbモードでは、最適化 Keilは、条件付きフラグをサポートするロード命令がないため、条件付きジャンプ命令を使用する必要があります。

Listing 1.126: 最適化 Keil 6/2013 (Thumbモード)

```
f PROC
; 入力値と10を比較
    CMP    r0,#0xa
; 同じなら、|L0.8|にジャンプ
    BEQ    |L0.8|
    ADR    r0,|L0.12| ; "it is not ten"
    BX     lr
|L0.8|
    ADR    r0,|L0.28| ; "it is ten"
    BX     lr
ENDP

|L0.12|
DCB      "it is not ten",0
|L0.28|
DCB      "it is ten",0
```

ARM64

ARM64の 最適化 GCC (Linaro) 4.9でも、条件付きジャンプが使用されます。

Listing 1.127: 最適化 GCC (Linaro) 4.9

```
f:
    cmp    x0, 10
    beq    .L3          ; 等しければ分岐
    adrp   x0, .LC1      ; "it is ten"
    add    x0, x0, :lo12:LC1
    ret

.L3:
    adrp   x0, .LC0      ; "it is not ten"
    add    x0, x0, :lo12:LC0
    ret

.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
```

これは、ARM64には32ビットARMモードの ADRcc やx86の CMOVcc などの条件フラグを伴った単純なロード命令がないためです。

しかし、「Conditional SElect」命令(CSEL)[*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)p390, C5.5] を使用していますが、GCC 4.9ではこのようなコードの中で使用するには十分スマートではないようです。

MIPS

残念ながら、MIPS用のGCC 4.4.5はそれほどスマートではありません。

Listing 1.128: 最適化 GCC 4.4.5 (アセンブリ出力)

```

$LC0:
    .ascii "it is not ten\000"
$LC1:
    .ascii "it is ten\000"
f:
    li      $2,10                # 0xa
; $a0と10を比較し、等しければ分岐
    beq     $4,$2,$L2
    nop ; branch delay slot

; "it is not ten" 文字列へのアドレスを $v0に残しつつリターン
    lui     $2,%hi($LC0)
    j       $31
    addiu   $2,$2,%lo($LC0)

$L2:
; "it is ten" 文字列へのアドレスを $v0に残しつつリターン
    lui     $2,%hi($LC1)
    j       $31
    addiu   $2,$2,%lo($LC1)

```

if/else の方法で書き直しましょう

```

const char* f (int a)
{
    if (a==10)
        return "it is ten";
    else
        return "it is not ten";
};

```

興味深いことに、x86用のGCC 4.8の最適化は、この場合に `CMOVcc` を使用することもできました。

Listing 1.129: 最適化 GCC 4.8

```

.LC0:
    .string "it is ten"
.LC1:
    .string "it is not ten"
f:
.LFB0:
; 入力値と10を比較
    cmp     DWORD PTR [esp+4], 10
    mov     edx, OFFSET FLAT:.LC1 ; "it is not ten"
    mov     eax, OFFSET FLAT:.LC0 ; "it is ten"
; 比較結果が同じでなければ、EDXの値をEAXにコピー
; そうでなければ、何もしない
    cmovne  eax, edx
    ret

```


ARMモードの最適化 Keilでは、リスト1.125 と同じコードが生成されます。

しかし、MSVC 2012の最適化は（まだ）あまり良くありません。

結論

コンパイラを最適化するとどうして条件付きジャンプを取り除こうとするのでしょうか？
以下を読んでください：?? on page ??

第1.14.4節最小値と最大値の取得

32-bit

```
int my_max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
};

int my_min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
};
```

Listing 1.130: 非最適化 MSVC 2013

```
_a$ = 8
_b$ = 12
_my_min PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
; AとBを比較
    cmp     eax, DWORD PTR _b$[ebp]
; AがB以上の場合にジャンプする
    jge     SHORT $LN2@my_min
; それ以外ではAをEAXにリロードして終了する
    mov     eax, DWORD PTR _a$[ebp]
    jmp     SHORT $LN3@my_min
    jmp     SHORT $LN3@my_min ; これは冗長なJMP命令
$LN2@my_min:
; Bをリターン
    mov     eax, DWORD PTR _b$[ebp]
$LN3@my_min:
    pop     ebp
    ret     0
_my_min ENDP
```

```

_a$ = 8
_b$ = 12
_my_max PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
; AとBを比較
    cmp     eax, DWORD PTR _b$[ebp]
; AがB以下の場合ジャンプする
    jle     SHORT $LN2@my_max
; それ以外ではAをEAXにリロードして終了する
    mov     eax, DWORD PTR _a$[ebp]
    jmp     SHORT $LN3@my_max
    jmp     SHORT $LN3@my_max ; これは冗長なJMP命令
$LN2@my_max:
; Bをリターン
    mov     eax, DWORD PTR _b$[ebp]
$LN3@my_max:
    pop     ebp
    ret     0
_my_max ENDP

```

これらの2つの機能は条件ジャンプ命令でのみ異なります。最初の命令では JGE (「Jump if Greater or Equal」) が使用され、2番目の場合は JLE (「Jump if Less or Equal」) が使用されます。

各関数には不必要な JMP 命令が1つありますが、おそらく誤って残っています。

分岐

ThumbモードのARMは、x86コードを思い起こします。

Listing 1.131: 最適化 Keil 6/2013 (Thumbモード)

```

my_max PROC
; R0=A
; R1=B
; AとBを比較
    CMP     r0,r1
; AがBより大きければ分岐
    BGT     |L0.6|
; それ以外 (A<=B) の場合は、R1(B) をリターン
    MOVS     r0,r1
|L0.6|
; リターン
    BX      lr
    ENDP

my_min PROC
; R0=A
; R1=B
; AとBを比較

```

```

        CMP        r0,r1
; AがBより小さければ分岐
        BLT        |L0.14|
; それ以外 (A>=B) の場合は、R1(B) をリターン
        MOVS       r0,r1
|L0.14|
; リターン
        BX         lr
        ENDP

```

関数は分岐命令が異なります。BGT と BLT です。ARMモードでは条件付きの接尾辞を使用することができるため、コードは短くなります。

MOVcc は、条件が満たされた場合にのみ実行されます。

Listing 1.132: 最適化 Keil 6/2013 (ARMモード)

```

my_max PROC
; R0=A
; R1=B
; AとBを比較
        CMP        r0,r1
; BをR0に入れて、AではなくBをリターン
; A<=Bのときにのみ、この命令は実行されます (つまり、LE - Less or Equal)
; 命令が実行されない場合 (A>Bのとき)、AはR0レジスタにあります。
        MOVLE      r0,r1
        BX         lr
        ENDP

my_min PROC
; R0=A
; R1=B
; AとBを比較
        CMP        r0,r1
; BをR0に入れて、AではなくBをリターン
; A>=Bのときにのみ、この命令は実行されます (つまり、GE - Greater or Equal)
; 命令が実行されない場合 (A<Bのとき)、AはR0レジスタにあります。
        MOVGE      r0,r1
        BX         lr
        ENDP

```

最適化 GCC 4.8.1とMSVC 2013の最適化では、ARMの CMOVcc に似た MOVcc 命令を使用できます。

Listing 1.133: 最適化 MSVC 2013

```

my_max:
        mov        edx, DWORD PTR [esp+4]
        mov        eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; AとBを比較
        cmp        edx, eax
; A>=Bなら、Aの値をEAXにロード

```

```

; それ以外 (A<B) の場合は、アイドル命令
    cmovge    eax, edx
    ret

my_min:
    mov     edx, DWORD PTR [esp+4]
    mov     eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; AとBを比較
    cmp     edx, eax
; A<=Bなら、Aの値をEAXにロード
; それ以外 (A>B) の場合は、アイドル命令
    cmovle    eax, edx
    ret

```

64-bit

```

#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
    if (a>b)
        return a;
    else
        return b;
};

int64_t my_min(int64_t a, int64_t b)
{
    if (a<b)
        return a;
    else
        return b;
};

```

いくつかの不要な値のシャッフルがありますが、コードは理解できます。

Listing 1.134: 非最適化 GCC 4.9.1 ARM64

```

my_max:
    sub     sp, sp, #16
    str     x0, [sp,8]
    str     x1, [sp]
    ldr     x1, [sp,8]
    ldr     x0, [sp]
    cmp     x1, x0
    ble     .L2
    ldr     x0, [sp,8]
    b       .L3
.L2:
    ldr     x0, [sp]

```

```

.L3:
    add    sp, sp, 16
    ret

my_min:
    sub    sp, sp, #16
    str    x0, [sp,8]
    str    x1, [sp]
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    cmp    x1, x0
    bge    .L5
    ldr    x0, [sp,8]
    b      .L6

.L5:
    ldr    x0, [sp]

.L6:
    add    sp, sp, 16
    ret

```

分岐なし

スタックから関数の引数をロードする必要はありません。レジスタにすでに入っています。

Listing 1.135: 最適化 GCC 4.9.1 x64

```

my_max:
; RDI=A
; RSI=B
; AとBを比較
    cmp    rdi, rsi
; Bを戻り値としてRAXにコピー
    mov    rax, rsi
; A>=Bの場合、A(RDI) を戻り値としてRAXにコピー
; それ以外 (A<B) では、アイドル命令
    cmovge rax, rdi
    ret

my_min:
; RDI=A
; RSI=B
; AとBを比較
    cmp    rdi, rsi
; Bを戻り値としてRAXにコピー
    mov    rax, rsi
; A<=Bの場合、A(RDI) を戻り値としてRAXにコピー
; それ以外 (A>B) では、アイドル命令
    cmovle rax, rdi
    ret

```

MSVC 2013はほぼ同じです。

ARM64にはARMの MOVcc またはx86の CMOVcc と同じように機能する CSEL 命令がありますが、その名前は「Conditional SElect」とは異なります。

Listing 1.136: 最適化 GCC 4.9.1 ARM64

```
my_max:
; X0=A
; X1=B
; AとBを比較
    cmp    x0, x1
; X0>=X1 または A>=B (Greater or Equal) の場合、X0(A) を選択する
; A<Bの場合、X1 (B) を選択する
    csel   x0, x0, x1, ge
    ret

my_min:
; X0=A
; X1=B
; AとBを比較
    cmp    x0, x1
; X0<=X1 または A<=B (Less or Equal) の場合、X0(A) を選択する
; A>Bの場合、X1 (B) を選択する
    csel   x0, x0, x1, le
    ret
```

MIPS

残念ながら、MIPS用のGCC 4.4.5はあまり良くありません。

Listing 1.137: 最適化 GCC 4.4.5 (IDA)

```
my_max:
; $a1<$a0なら、$v1に1を設定し、それ以外 ($a1>$a0) ではクリアする
    slt    $v1, $a1, $a0
; $v1が0(または $a1>$a0) ならジャンプ
    beqz   $v1, locret_10
; これは分岐遅延スロットです
; 分岐が実行された場合に、$a1を $v0にコピー
    move   $v0, $a1
; 分岐は実行されず、$a0を $v0にコピー
    move   $v0, $a0

locret_10:
    jr     $ra
    or     $at, $zero ; 分岐遅延スロット、NOP

; min() 関数は同じですが、SLT命令の入力オペランドはスワップされます
my_min:
    slt    $v1, $a0, $a1
    beqz   $v1, locret_28
    move   $v0, $a1
    move   $v0, $a0

locret_28:
```

```

jr      $ra
or      $at, $zero ; branch delay slot, NOP

```

分岐遅延スロットを忘れないでください。最初の MOVE は BEQZ の前に実行され、2番目の MOVE は分岐が実行されなかった場合にのみ実行されます。

第1.14.5節結論

x86

条件付きジャンプの基本骨格は次のとおりです。

Listing 1.138: x86

```

CMP register, register/value
Jcc true ; cc=condition code
false:
;... 比較結果が偽の場合に実行されるコード...
JMP exit
true:
;... 比較結果が真の場合に実行されるコード...
exit:

```

ARM

Listing 1.139: ARM

```

CMP register, register/value
Bcc true ; cc=condition code
false:
;... 比較結果が偽の場合に実行されるコード...
JMP exit
true:
;... 比較結果が真の場合に実行されるコード...
exit:

```

MIPS

Listing 1.140: Check for zero

```

BEQZ REG, label
...

```

Listing 1.141: Check for less than zero using pseudoinstruction

```

BLTZ REG, label
...

```

Listing 1.142: Check for equal values

```

BEQ REG1, REG2, label
...

```

Listing 1.143: Check for non-equal values

```
BNE REG1, REG2, label
...
```

Listing 1.144: Check for less than (signed)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Listing 1.145: Check for less than (unsigned)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

Branchless

条件文の本体が非常に短い場合は、ARMの MOVcc (ARMモードの場合)、ARM64の場合は CSEL、x86の場合は CMOVcc の条件付き移動命令を使用できます。

ARM

命令によっては、ARMモードで条件付き接尾辞を使用することもできます。

Listing 1.146: ARM (ARMモード)

```
CMP register, register/value
instr1_cc ; 条件コードが真の場合、何らかの命令が実行されます
instr2_cc ; 他の条件コードが真の場合、他の命令が実行されます
;... etc...
```

もちろん、CPUフラグがいずれかで変更されない限り、条件付きコードの接尾辞付き命令の数に制限はありません。

Thumbモードには IT 命令があり、次の4つの命令に条件付きサフィックスを追加できます。詳しくは、[1.19.7 on page 316](#)を参照してください。

Listing 1.147: ARM (Thumbモード)

```
CMP register, register/value
ITEEE EQ ; 接尾辞を設定します: if-then-else-else-else
instr1 ; 条件が真であれば命令が実行されます
instr2 ; 条件が偽であれば命令が実行されます
instr3 ; 条件が偽であれば命令が実行されます
instr4 ; 条件が偽であれば命令が実行されます
```

第1.14.6節練習問題

(ARM64) すべての条件付きジャンプ命令を削除し、CSEL 命令を使用して、リスト[1.127](#)のコードを書き直してみてください。

第1.15節switch()/case/default

第1.15.1節小さな数のcase

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};
```

x86

非最適化 **MSVC**

結果 (MSVC 2010):

Listing 1.148: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 0
    je      SHORT $LN4@f
    cmp     DWORD PTR tv64[ebp], 1
    je      SHORT $LN3@f
    cmp     DWORD PTR tv64[ebp], 2
    je      SHORT $LN2@f
    jmp     SHORT $LN1@f
$LN4@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN3@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
```

```

    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN2@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN1@f:
    push    OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call    _printf
    add     esp, 4
$LN7@f:
    mov     esp, ebp
    pop     ebp
    ret     0
_f        ENDP

```

実際、switch() でいくつかのcaseを持つ私たちの関数は、この構造に似ています。

```

void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};

```

いくつかのcaseでswitch() を使用する場合、ソースコード内の実際のswitch() か、単にif文の組であるかどうかを確認することは不可能です。

これはswitch() が多段にネストされたif文との糖衣構文のようなものであることを意味します。

コンパイラが入力変数 *a* を一時的なローカル変数 tv64 に移動することを除いて、生成されたコードには特に新しいことはありません。⁹¹

これをGCC 4.4.1でコンパイルすると、最大限の最適化 (-O3 option) を有効にしてもほぼ同じ結果になります。

最適化 MSVC

では、MSVC (/Ox) の最適化を有効にしましょう : cl 1.c /Fa1.asm /Ox

Listing 1.149: MSVC

⁹¹ スタック内のローカル変数には接頭辞 tv が付きます。MSVCが内部変数として使用するために命名しています。

```

_a$ = 8 ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    sub     eax, 0
    je      SHORT $LN4@f
    sub     eax, 1
    je      SHORT $LN3@f
    sub     eax, 1
    je      SHORT $LN2@f
    mov     DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH,
    00H
    jmp     _printf
$LN2@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp     _printf
$LN3@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp     _printf
$LN4@f:
    mov     DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00H
    jmp     _printf
_f ENDP

```

ここで、汚いハックを見ることができます。

最初に、*a* の値を EAX に置き、0を引きます。EAXの値が0かどうかを確認するために行われますが、そうであれば、ZF フラグがセットされます（例えば、0からの減算は0）最初の条件ジャンプ JE (*Jump if Equal* またはあ同義語 JZ —*Jump if Zero*) は実行され、制御フローは \$LN4@f ラベルに渡されます。ここでは、'zero' メッセージが出力されます。最初のジャンプが実行されない場合は、入力値から1が減算され、結果が0の場合、対応するジャンプが実行されます。

また、ジャンプが全く実行されない場合、制御フローは文字列引数'something unknown' を printf() に渡します。

次に、文字列ポインタが *a* 変数に置かれ、printf() が CALL ではなく JMP を介して呼び出されます。簡単に説明するとこうなります：caller は値をスタックにプッシュし、CALL 経由で関数を呼び出します。CALL 自体は戻りアドレス (RA) をスタックにプッシュし、関数アドレスへの無条件ジャンプを行います。スタックポインタを移動させる命令が含まれていないため、任意の実行時点での関数は、次のスタックレイアウトを持ちます。

- ESP—points to RA
- ESP+4—points to the *a* variable

反対に、printf() をここで呼び出さなければならないときは、文字列を指し示す必要がある最初の printf() 引数を除いて、全く同じスタックレイアウトが必要です。それが私たちのコードがすることです。

ファクションの最初の引数を文字列のアドレスに置き換え、関数 f() を直接呼び出せずに直接 printf() を呼び出すかのように、printf() にジャンプします。printf() は文字列を stdout に出力し、RET 命令を実行します。スタックからRAを取り出し、制御フローは f() ではなく f() 関数の終りをバイパスして、f() の caller です。

`printf()` はすべての場合に `f()` 関数の終わりで右に呼ばれるので、これはすべて可能です。ある意味では、`longjmp()`⁹² 関数に似ています。そしてもちろん、それはスピードのためにすべて行われます。

ARMコンパイラと同様のケースは、「`printf()` 引数を取って」セクションに記載されています。こちら：[\(1.8.2 on page 68\)](#)

⁹²[wikipedia](#)

OllyDbg

この例は扱いにくいので、OllyDbg でトレースしてみましょう。

OllyDbg はそのようなswitch() 構文を検出することができ、有用なコメントを追加することができます。EAX の値は最初は2で、それは関数への入力値です：

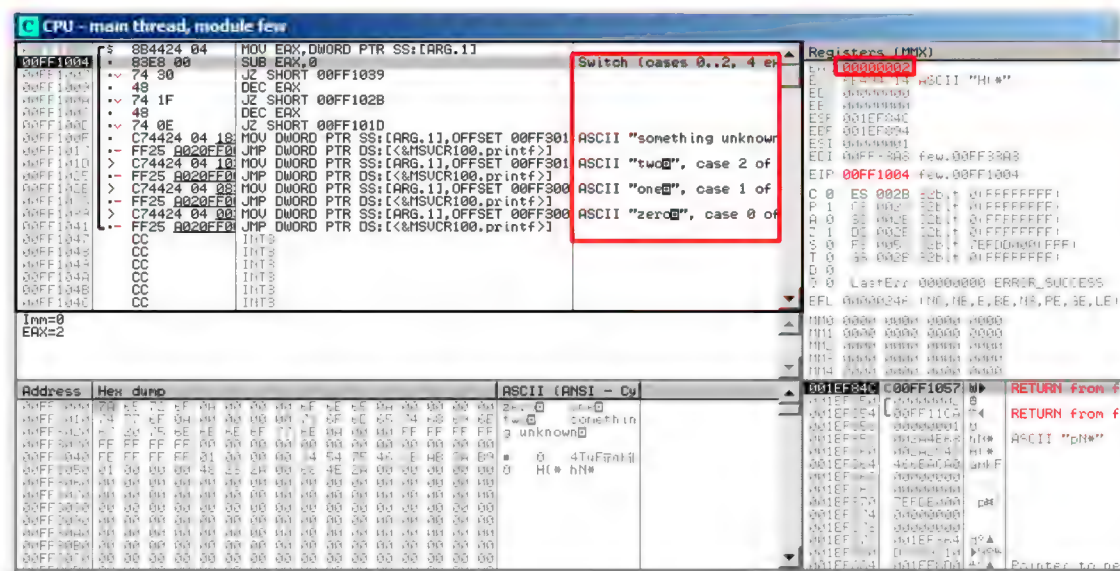


図 1.42: OllyDbg: EAX は最初の（そして唯一の）関数への引数を含んでいます

0は EAX から2を引いた値です。もちろん、EAX にはまだ2が入っています。しかし、ZF フラグは0になり、結果の値がゼロでないことを示します。

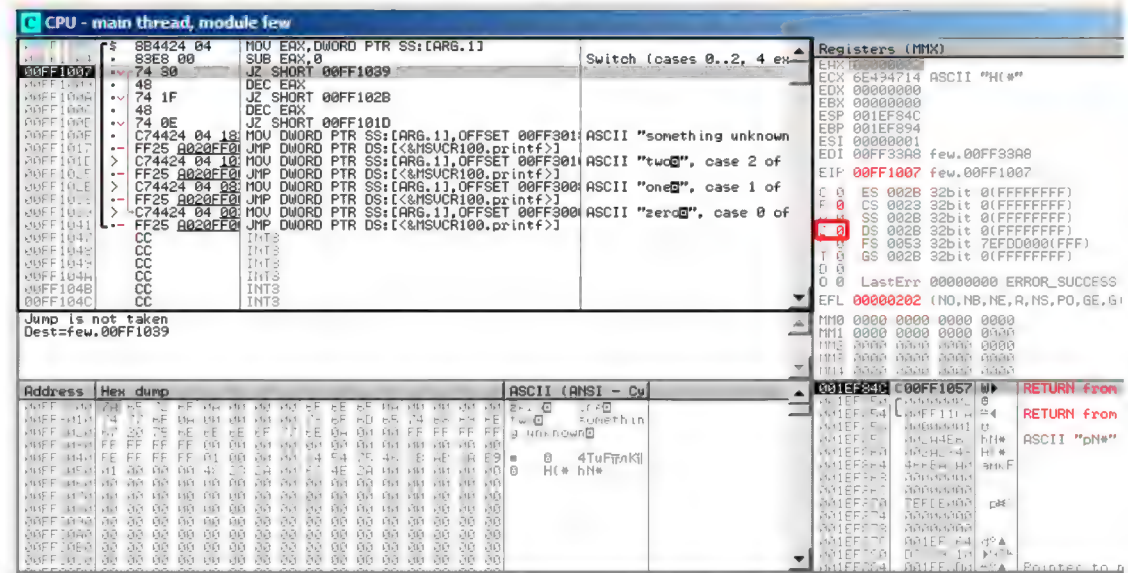


図 1.43: OllyDbg: SUB の実行

DEC が実行され、EAX には1が入ります。しかし1はゼロではないので、ZF フラグはまだ0です：

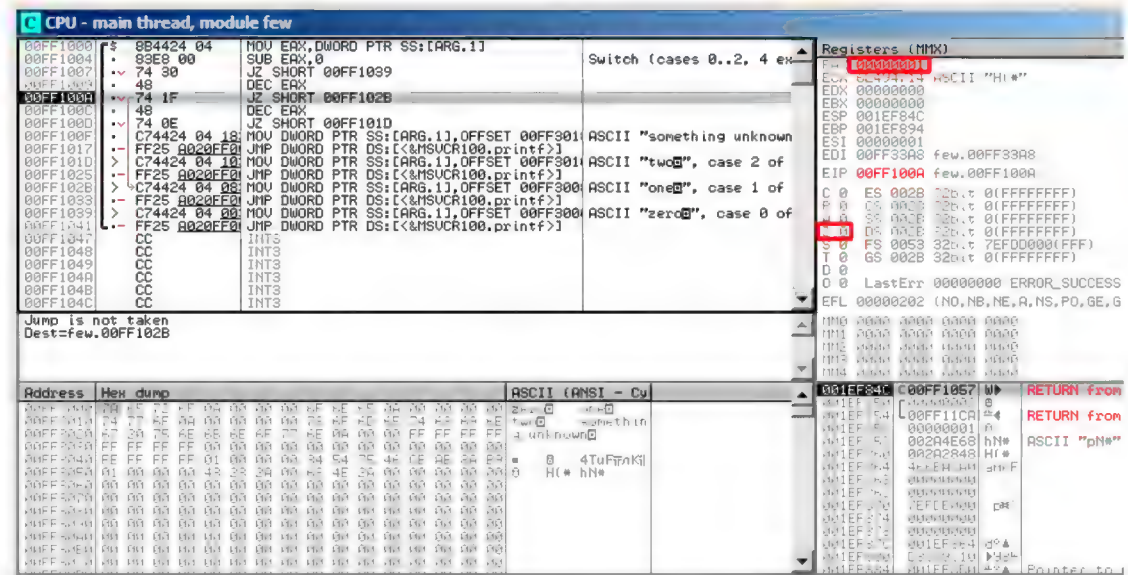


図 1.44: OllyDbg: 最初の DEC 実行

次の DEC が実行されます。EAX は最終的に0になり、結果がゼロであるため ZF フラグが設定されます。

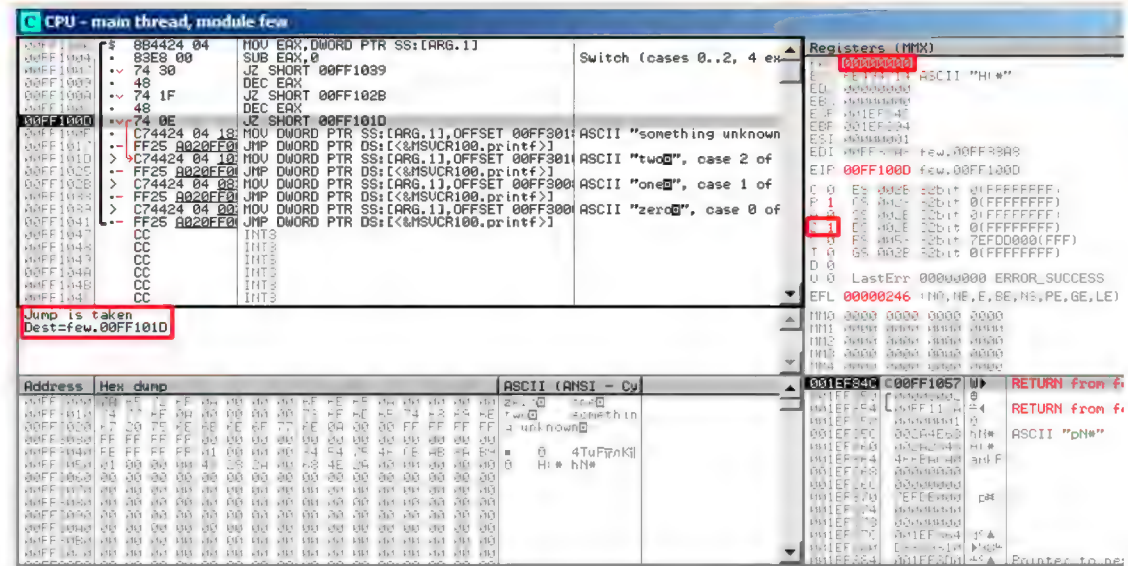


図 1.45: OllyDbg: 2回目の DEC 実行

OllyDbg は、このジャンプが今行われることを示しています。

「two」という文字列へのポインタが今スタックに書き込まれます：

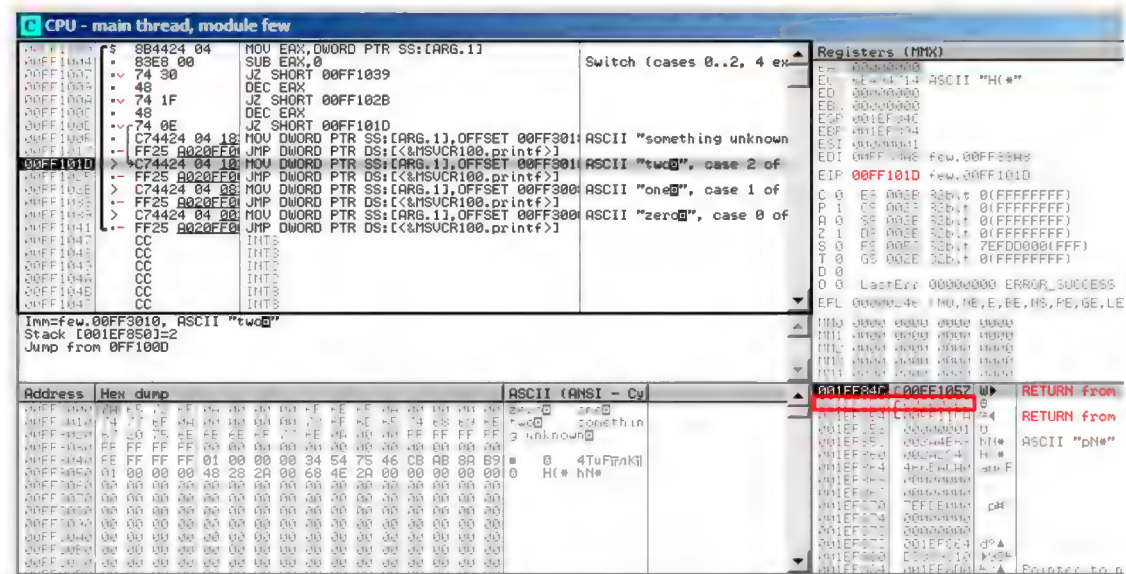


図 1.46: OllyDbg: 文字列へのポインタは、最初の引数の場所書き込まれる

注意：関数の現在の引数は2であり、2はスタックに 0x001EF850 のアドレスにあります。

これが printf() の最後の命令です。

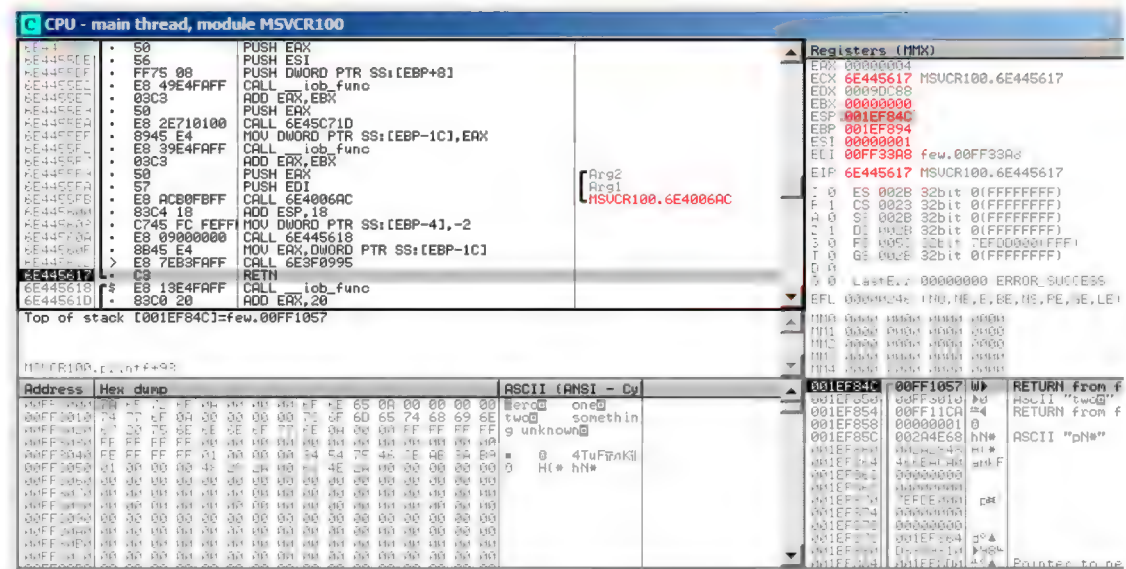


図 1.48: OllyDbg: MSVCRT100.DLLの printf() の最後の命令

文字列「two」はコンソールウィンドウに表示されます。

F7またはF8を押して(ステップオーバー)リターンすると...f()ではなく、main()にいきます。

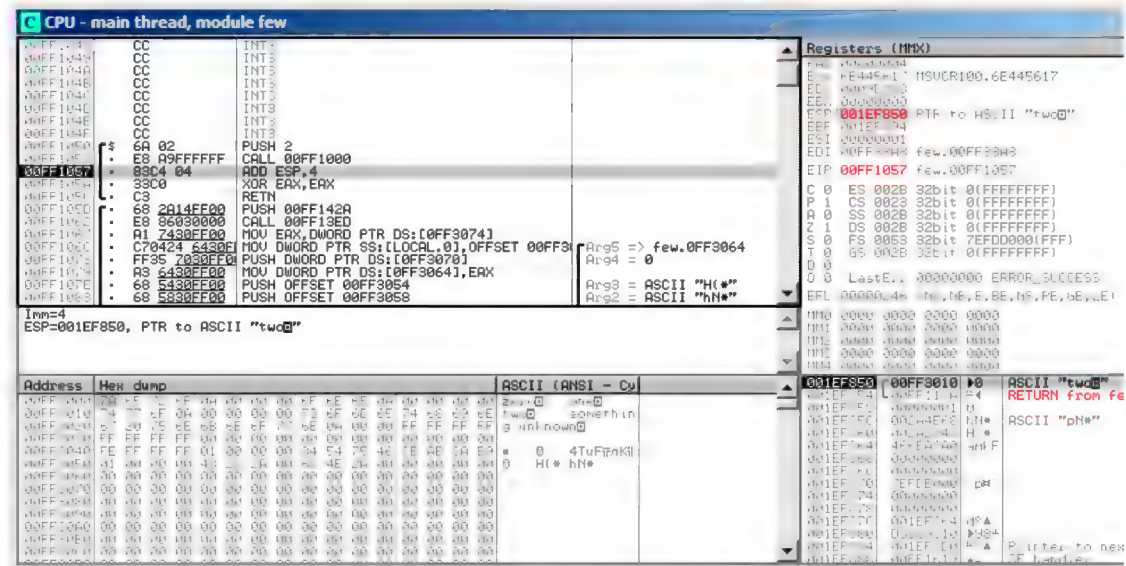


図 1.49: OllyDbg: main() へのリターン

はい、printf() の中心から main() に直接ジャンプしました。なぜならスタックのRAは f() ではなく、main() の場所を指しているからです。CALL 0x00FF1000 は f() を呼び出した実際の命令です。

ARM: 最適化 Keil 6/2013 (ARMモード)

.text:0000014C	f1:
.text:0000014C 00 00 50 E3	CMP R0, #0
.text:00000150 13 0E 8F 02	ADREQ R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A	BEQ loc_170
.text:00000158 01 00 50 E3	CMP R0, #1
.text:0000015C 4B 0F 8F 02	ADREQ R0, aOne ; "one\n"
.text:00000160 02 00 00 0A	BEQ loc_170
.text:00000164 02 00 50 E3	CMP R0, #2
.text:00000168 4A 0F 8F 12	ADRNE R0, aSomethingUnkno ; "something unknown\n"
.text:0000016C 4E 0F 8F 02	ADREQ R0, aTwo ; "two\n"
.text:00000170	
.text:00000170	loc_170: ; CODE XREF: f1+8
.text:00000170	; f1+14
.text:00000170 78 18 00 EA	B __2printf

繰り返しますが、このコードを調べることで、元のソースコードのswitch() か単なるif() 文の集合かどうかはわかりません。

とにかく、ここでは、 $R0 = 0$ の場合にのみトリガされる ADREQ (*Equal*) のような述語命令を再度参照し、文字列 «zero\n» を $R0$ にコピーします。 $R0 = 0$ の場合、次の命令 BEQ は制御フローを loc_170 にリダイレクトします。

巧妙な読者は、 $R0$ レジスタに既に値を埋め込んでいるので、BEQ が正しくトリガされるかどうかを尋ねるかもしれません。

はい、BEQ は CMP 命令で設定されたフラグをチェックし、ADREQ はフラグをまったく変更しません。

命令の残りの部分は既に慣れ親しんでいます。最後に printf() を1回呼び出すだけですが、ここではこのトリック (1.8.2 on page 68) を既に調べています。最後に printf() には3つのパスがあります。

$a = 2$ かどうかを確認するには、最後の命令 CMP $R0$, #2 が必要です。

それが真でない場合、ADRNE は a がすでに0に等しいとチェックされているので、«something unknown\n» 文字列へのポインタを $R0$ にロードしますまたは1であり、この時点で a 変数がこれらの数値と等しくないことがわかります。 $R0 = 2$ の場合、文字列 «two\n» へのポインタは ADREQ によって $R0$ にロードされます。

ARM: 最適化 Keil 6/2013 (Thumbモード)

```
.text:000000D4          f1:
.text:000000D4 10 B5      PUSH    {R4,LR}
.text:000000D6 00 28      CMP     R0, #0
.text:000000D8 05 D0      BEQ     zero_case
.text:000000DA 01 28      CMP     R0, #1
.text:000000DC 05 D0      BEQ     one_case
.text:000000DE 02 28      CMP     R0, #2
.text:000000E0 05 D0      BEQ     two_case
.text:000000E2 91 A0      ADR     R0, aSomethingUnkno ; "something
unknown\n"
.text:000000E4 04 E0      B       default_case

.text:000000E6          zero_case: ; CODE XREF: f1+4
.text:000000E6 95 A0      ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0      B       default_case

.text:000000EA          one_case: ; CODE XREF: f1+8
.text:000000EA 96 A0      ADR     R0, aOne ; "one\n"
.text:000000EC 00 E0      B       default_case

.text:000000EE          two_case: ; CODE XREF: f1+C
.text:000000EE 97 A0      ADR     R0, aTwo ; "two\n"
.text:000000F0          default_case ; CODE XREF: f1+10
.text:000000F0          ; f1+14
.text:000000F0 06 F0 7E F8  BL     __2printf
.text:000000F4 10 BD      POP     {R4,PC}
```

既に言及したように、Thumbモードのほとんどの命令に条件付き述語を追加することではできないため、このThumbコードは、わかりやすいx86 CISCスタイルコードと多少似ています。

ARM64: 非最適化 GCC (Linaro) 4.9

```

.LC12:
    .string "zero"
.LC13:
    .string "one"
.LC14:
    .string "two"
.LC15:
    .string "something unknown"
f12:
    stp    x29, x30, [sp, -32]!
    add    x29, sp, 0
    str    w0, [x29,28]
    ldr    w0, [x29,28]
    cmp    w0, 1
    beq    .L34
    cmp    w0, 2
    beq    .L35
    cmp    w0, wzr
    bne    .L38                ; デフォルトのラベルにジャンプ
    adrp   x0, .LC12            ; "zero"
    add    x0, x0, :lo12:.LC12
    bl     puts
    b      .L32
.L34:
    adrp   x0, .LC13            ; "one"
    add    x0, x0, :lo12:.LC13
    bl     puts
    b      .L32
.L35:
    adrp   x0, .LC14            ; "two"
    add    x0, x0, :lo12:.LC14
    bl     puts
    b      .L32
.L38:
    adrp   x0, .LC15            ; "something unknown"
    add    x0, x0, :lo12:.LC15
    bl     puts
    nop
.L32:
    ldp    x29, x30, [sp], 32
    ret

```

入力値のタイプは *int* なので、X0 レジスタ全体ではなくレジスタ W0 が使用されます。

文字列ポインタは ADRP/ADD 命令ペアを使用して「ハローワールド!」の例 [1.5.3 on page 31](#)と同じように puts() に渡されます。

ARM64: 最適化 GCC (Linaro) 4.9

```

f12:
    cmp    w0, 1

```



```

        beq      .L31
        cmp      w0, 2
        beq      .L32
        cbz      w0, .L35
; デフォルトの場合
        adrp     x0, .LC15          ; "something unknown"
        add      x0, x0, :lo12:.LC15
        b        puts
.L35:
        adrp     x0, .LC12          ; "zero"
        add      x0, x0, :lo12:.LC12
        b        puts
.L32:
        adrp     x0, .LC14          ; "two"
        add      x0, x0, :lo12:.LC14
        b        puts
.L31:
        adrp     x0, .LC13          ; "one"
        add      x0, x0, :lo12:.LC13
        b        puts

```

より最適化されたコード。R0 がゼロの場合、CBZ (*Compare and Branch on Zero*) 命令はジャンプします。また、[1.15.1 on page 190](#)の前に説明したように、puts() を呼び出す代わりに直接ジャンプすることもできます。

MIPS

Listing 1.150: 最適化 GCC 4.4.5 (IDA)

```

f:
        lui      $gp, (__gnu_local_gp >> 16)
; 1か?
        li       $v0, 1
        beq      $a0, $v0, loc_60
        la       $gp, (__gnu_local_gp & 0xFFFF) ; 分岐遅延スロット
; 2か?
        li       $v0, 2
        beq      $a0, $v0, loc_4C
        or       $at, $zero ; 分岐遅延スロット、NOP
; 0と等しくなければジャンプ
        bnez     $a0, loc_38
        or       $at, $zero ; 分岐遅延スロット、NOP
; 0の場合
        lui      $a0, ($LC0 >> 16) # "zero"
        lw       $t9, (puts & 0xFFFF)($gp)
        or       $at, $zero ; 分岐遅延スロット、NOP
        jr       $t9 ; 分岐遅延スロット、NOP
        la       $a0, ($LC0 & 0xFFFF) # "zero" ; 分岐遅延スロット

loc_38:
                                # CODE XREF: f+1C
        lui      $a0, ($LC3 >> 16) # "something unknown"
        lw       $t9, (puts & 0xFFFF)($gp)
        or       $at, $zero ; 分岐遅延スロット、NOP

```

スロット	jr	\$t9	
	la	\$a0, (\$LC3 & 0xFFFF)	# "something unknown" ; 分岐遅延
loc_4C:			
			# CODE XREF: f+14
	lui	\$a0, (\$LC2 >> 16)	# "two"
	lw	\$t9, (puts & 0xFFFF)(\$gp)	
	or	\$at, \$zero	; 分岐遅延スロット、NOP
	jr	\$t9	
	la	\$a0, (\$LC2 & 0xFFFF)	# "two" ; 分岐遅延スロット
loc_60:			
			# CODE XREF: f+8
	lui	\$a0, (\$LC1 >> 16)	# "one"
	lw	\$t9, (puts & 0xFFFF)(\$gp)	
	or	\$at, \$zero	; 分岐遅延スロット、NOP
	jr	\$t9	
	la	\$a0, (\$LC1 & 0xFFFF)	# "one" ; 分岐遅延スロット

関数は常に puts() を呼び出すことで終了するので、puts() (JR :「Jump Register」) へのジャンプは「jump and link」ではなく、ここにあります。私たちは以前これについて話しました：[1.15.1 on page 190](#)

LW 命令の後に NOP 命令も表示されることがよくあります。これは「load delay slot」: MIPSの別の *delay slot* です。

LW がメモリから値をロードする間に、LW の次の命令が実行されることがあります。

ただし、次の命令は LW の結果を使用してはなりません。

現代のMIPS CPUは、次の命令が LW の結果を使用するのを待つ機能を持っているので、これは幾分時代遅れですが、GCCは古いMIPS CPU用にNOPを追加します。一般に、無視することができます。

結論

ほとんどの場合switch () はif / else構造と区別できません：リスト[1.15.1](#).

第1.15.2節A lot of cases

switch() ステートメントに大量のケースが含まれている場合、コンパイラが多くの JE/JNE 命令で大きすぎるコードを出力することはあまり便利ではありません。

```
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        case 3: printf ("three\n"); break;
        case 4: printf ("four\n"); break;
    }
}
```



```

        default: printf ("something unknown\n"); break;
    };
};

int main()
{
    f (2); // test
};

```

x86

非最適化 MSVC

We get (MSVC 2010):

Listing 1.151: MSVC 2010

```

tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 4
    ja      SHORT $LN1@f
    mov     ecx, DWORD PTR tv64[ebp]
    jmp     DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN5@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN4@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN3@f:
    push    OFFSET $SG745 ; 'three', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN2@f:
    push    OFFSET $SG747 ; 'four', 0aH, 00H
    call    _printf

```

```

    add    esp, 4
    jmp    SHORT $LN9@f
$LN1@f:
    push   OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call   _printf
    add    esp, 4
$LN9@f:
    mov    esp, ebp
    pop    ebp
    ret    0
    npad   2 ; 次のラベルにアラインメントする
$LN11@f:
    DD     $LN6@f ; 0
    DD     $LN5@f ; 1
    DD     $LN4@f ; 2
    DD     $LN3@f ; 3
    DD     $LN2@f ; 4
_f       ENDP

```

ここでは、さまざまな引数を持つ printf() 呼び出しのセットを見ていきます。すべては、プロセスのメモリだけでなく、コンパイラによって割り当てられた内部シンボリックラベルも持っています。これらのラベルはすべて \$LN11@f 内部テーブルにも記載されています。

関数の開始時に、*a* が4より大きい場合、制御フローはラベル \$LN1@f に渡されます。引数 'something unknown' をとって printf() が呼び出されます。

しかし、*a* の値が4以下の場合、4を乗算して \$LN11@f テーブルアドレスで加算します。これはテーブル内のアドレスがどのように構築され、必要な要素を正確に指し示すものです。たとえば、*a* が2に等しいとしましょう。 $2 * 4 = 8$ (すべてのテーブル要素は 32ビットプロセスのアドレスなので、すべての要素が4バイト幅です)。\$LN11@f テーブルのアドレス + 8は \$LN4@f ラベルが格納されているテーブル要素です。JMP はテーブルから \$LN4@f アドレスを取り出し、それにジャンプします。

このテーブルはしばしば *jumptable* または *branch table*⁹³ と呼ばれます。

それから、対応する printf() は引数 'two' で呼び出されます。実際、`jmp DWORD PTR $LN11@f[ecx*4]` 命令は *jump to the DWORD that is stored at address \$LN11@f + ecx * 4*

`npad (?? on page ??)` は、4バイト (または16バイト) の境界に整列したアドレスに格納されるように次のラベルを整列するアセンブリ言語マクロです。これは、メモリバス、キャッシュメモリなどを介してメモリから32ビット値をフェッチすることができるため、プロセッサが整列している場合にはより効果的な方法でプロセッサに非常に適しています。

⁹³The whole method was once called *computed GOTO* in early versions of Fortran: [wikipedia](https://en.wikipedia.org/wiki/Computed_goto). Not quite relevant these days, but what a term!

OllyDbg

OllyDbg でこの例を試してみましょう。関数への入力値2が EAX にロードされます。

CPU - main thread, module lot

Address	Hex	dump	ASCII (ANSI - Cy)
010B1007	55		
010B1008	8BEC		
010B1009	51		
010B100A	8B45 08		
010B100B	8945 FC		
010B100C	837D FC 04		
010B100D	77 5A		
010B100E	8B4D FC		
010B100F	FF248D 7C100		
010B1010	68 00300B01		
010B1011	FF15 00200B00		
010B1012	83C4 04		
010B1013	EB 4E		
010B1014	68 00300B01		
010B1015	FF15 00200B00		
010B1016	83C4 04		
010B1017	EB 3E		
010B1018	68 00300B01		
010B1019	FF15 00200B00		
010B101A	83C4 04		
010B101B	EB 2E		

Registers (MMX)

Register	Value
EAX	00000002
ECX	6E494714 MSUCR100.__initenv
EDX	00000000
EBX	00000000
ESP	003CFD08
EBP	003CFD0C
ESI	00000001
EDI	010B3388 lot.010B3388
EIP	010B1007

Stack [003CFD08]=6E494714 (MSUCR100.__initenv)

Address	Hex	dump	ASCII (ANSI - Cy)
003CFD08	6E494714		
003CFD09	00000000		
003CFD0A	00000000		
003CFD0B	00000000		
003CFD0C	00000000		
003CFD0D	00000000		
003CFD0E	00000000		
003CFD0F	00000000		
003CFD10	00000000		
003CFD11	00000000		
003CFD12	00000000		
003CFD13	00000000		
003CFD14	00000000		
003CFD15	00000000		
003CFD16	00000000		
003CFD17	00000000		
003CFD18	00000000		
003CFD19	00000000		
003CFD1A	00000000		
003CFD1B	00000000		
003CFD1C	00000000		
003CFD1D	00000000		
003CFD1E	00000000		
003CFD1F	00000000		
003CFD20	00000000		
003CFD21	00000000		
003CFD22	00000000		
003CFD23	00000000		
003CFD24	00000000		
003CFD25	00000000		
003CFD26	00000000		
003CFD27	00000000		
003CFD28	00000000		
003CFD29	00000000		
003CFD2A	00000000		
003CFD2B	00000000		
003CFD2C	00000000		
003CFD2D	00000000		
003CFD2E	00000000		
003CFD2F	00000000		
003CFD30	00000000		
003CFD31	00000000		
003CFD32	00000000		
003CFD33	00000000		
003CFD34	00000000		
003CFD35	00000000		
003CFD36	00000000		
003CFD37	00000000		
003CFD38	00000000		
003CFD39	00000000		
003CFD3A	00000000		
003CFD3B	00000000		
003CFD3C	00000000		
003CFD3D	00000000		
003CFD3E	00000000		
003CFD3F	00000000		
003CFD40	00000000		
003CFD41	00000000		
003CFD42	00000000		
003CFD43	00000000		
003CFD44	00000000		
003CFD45	00000000		
003CFD46	00000000		
003CFD47	00000000		
003CFD48	00000000		
003CFD49	00000000		
003CFD4A	00000000		
003CFD4B	00000000		
003CFD4C	00000000		
003CFD4D	00000000		
003CFD4E	00000000		
003CFD4F	00000000		
003CFD50	00000000		
003CFD51	00000000		
003CFD52	00000000		
003CFD53	00000000		
003CFD54	00000000		
003CFD55	00000000		
003CFD56	00000000		
003CFD57	00000000		
003CFD58	00000000		
003CFD59	00000000		
003CFD5A	00000000		
003CFD5B	00000000		
003CFD5C	00000000		
003CFD5D	00000000		
003CFD5E	00000000		
003CFD5F	00000000		
003CFD60	00000000		
003CFD61	00000000		
003CFD62	00000000		
003CFD63	00000000		
003CFD64	00000000		
003CFD65	00000000		
003CFD66	00000000		
003CFD67	00000000		
003CFD68	00000000		
003CFD69	00000000		
003CFD6A	00000000		
003CFD6B	00000000		
003CFD6C	00000000		
003CFD6D	00000000		
003CFD6E	00000000		
003CFD6F	00000000		
003CFD70	00000000		
003CFD71	00000000		
003CFD72	00000000		
003CFD73	00000000		
003CFD74	00000000		
003CFD75	00000000		
003CFD76	00000000		
003CFD77	00000000		
003CFD78	00000000		
003CFD79	00000000		
003CFD7A	00000000		
003CFD7B	00000000		
003CFD7C	00000000		
003CFD7D	00000000		
003CFD7E	00000000		
003CFD7F	00000000		
003CFD80	00000000		
003CFD81	00000000		
003CFD82	00000000		
003CFD83	00000000		
003CFD84	00000000		
003CFD85	00000000		
003CFD86	00000000		
003CFD87	00000000		
003CFD88	00000000		
003CFD89	00000000		
003CFD8A	00000000		
003CFD8B	00000000		
003CFD8C	00000000		
003CFD8D	00000000		
003CFD8E	00000000		
003CFD8F	00000000		
003CFD90	00000000		
003CFD91	00000000		
003CFD92	00000000		
003CFD93	00000000		
003CFD94	00000000		
003CFD95	00000000		
003CFD96	00000000		
003CFD97	00000000		
003CFD98	00000000		
003CFD99	00000000		
003CFD9A	00000000		
003CFD9B	00000000		
003CFD9C	00000000		
003CFD9D	00000000		
003CFD9E	00000000		
003CFD9F	00000000		
003CFDA0	00000000		
003CFDA1	00000000		
003CFDA2	00000000		
003CFDA3	00000000		
003CFDA4	00000000		
003CFDA5	00000000		
003CFDA6	00000000		
003CFDA7	00000000		
003CFDA8	00000000		
003CFDA9	00000000		
003CFDAA	00000000		
003CFDAB	00000000		
003CFDAC	00000000		
003CFDAD	00000000		
003CFDAE	00000000		
003CFDAF	00000000		
003CFDB0	00000000		
003CFDB1	00000000		
003CFDB2	00000000		
003CFDB3	00000000		
003CFDB4	00000000		
003CFDB5	00000000		
003CFDB6	00000000		
003CFDB7	00000000		
003CFDB8	00000000		
003CFDB9	00000000		
003CFDBA	00000000		
003CFDBB	00000000		
003CFDBC	00000000		
003CFDBD	00000000		
003CFDBE	00000000		
003CFDBF	00000000		
003CFDC0	00000000		
003CFDC1	00000000		
003CFDC2	00000000		
003CFDC3	00000000		
003CFDC4	00000000		
003CFDC5	00000000		
003CFDC6	00000000		
003CFDC7	00000000		
003CFDC8	00000000		
003CFDC9	00000000		
003CFDD0	00000000		
003CFDD1	00000000		
003CFDD2	00000000		
003CFDD3	00000000		
003CFDD4	00000000		
003CFDD5	00000000		
003CFDD6	00000000		
003CFDD7	00000000		
003CFDD8	00000000		
003CFDD9	00000000		
003CFDDA	00000000		
003CFDDB	00000000		
003CFDDC	00000000		
003CFDDD	00000000		
003CFDDE	00000000		
003CFDDF	00000000		
003CFDE0	00000000		
003CFDE1	00000000		
003CFDE2	00000000		
003CFDE3	00000000		
003CFDE4	00000000		
003CFDE5	00000000		
003CFDE6	00000000		
003CFDE7	00000000		
003CFDE8	00000000		
003CFDE9	00000000		
003CFDEA	00000000		
003CFDEB	00000000		
003CFDEC	00000000		
003CFDED	00000000		
003CFDEE	00000000		
003CFDEF	00000000		
003CFDF0	00000000		
003CFDF1	00000000		
003CFDF2	00000000		
003CFDF3	00000000		
003CFDF4	00000000		
003CFDF5	00000000		
003CFDF6	00000000		
003CFDF7	00000000		
003CFDF8	00000000		
003CFDF9	00000000		
003CFDFA	00000000		
003CFDFB	00000000		
003CFDFC	00000000		
003CFDFD	00000000		
003CFDFE	00000000		
003CFDFF	00000000		
003CFD00	00000000		
003CFD01	00000000		
003CFD02	00000000		
003CFD03	00000000		
003CFD04	00000000		
003CFD05	00000000		
003CFD06	00000000		
003CFD07	00000000		
003CFD08	00000000		
003CFD09	00000000		
003CFD0A	00000000		
003CFD0B	00000000		
003CFD0C	00000000		
003CFD0D	00000000		
003CFD0E	00000000		
003CFD0F	00000000		
003CFD10	00000000		
003CFD11	00000000		
003CFD12	00000000		
003CFD13	00000000		
003CFD14	00000000		
003CFD15	00000000		
003CFD16	00000000		
003CFD17	00000000		
003CFD18	00000000		
003CFD19	00000000		
003CFD1A	00000000		
003CFD1B	00000000		
003CFD1C	00000000		
003CFD1D	00000000		
003CFD1E	00000000		
003CFD1F	00000000		
003CFD20	00000000		
003CFD21	00000000		
003CFD22	00000000		
003CFD23	00000000		
003CFD24	00000000		
003CFD25	00000000		
003CFD26	00000000		
003CFD27	00000000		
003CFD28	00000000		
003CFD29	00000000		
003CFD2A	00000000		
003CFD2B	00000000		
003CFD2C	00000000		
003CFD2D	00000000		
003CFD2E	00		

入力値が4より大きいかチェックされます。そうでなければ、「default」ジャンプは実行されません。

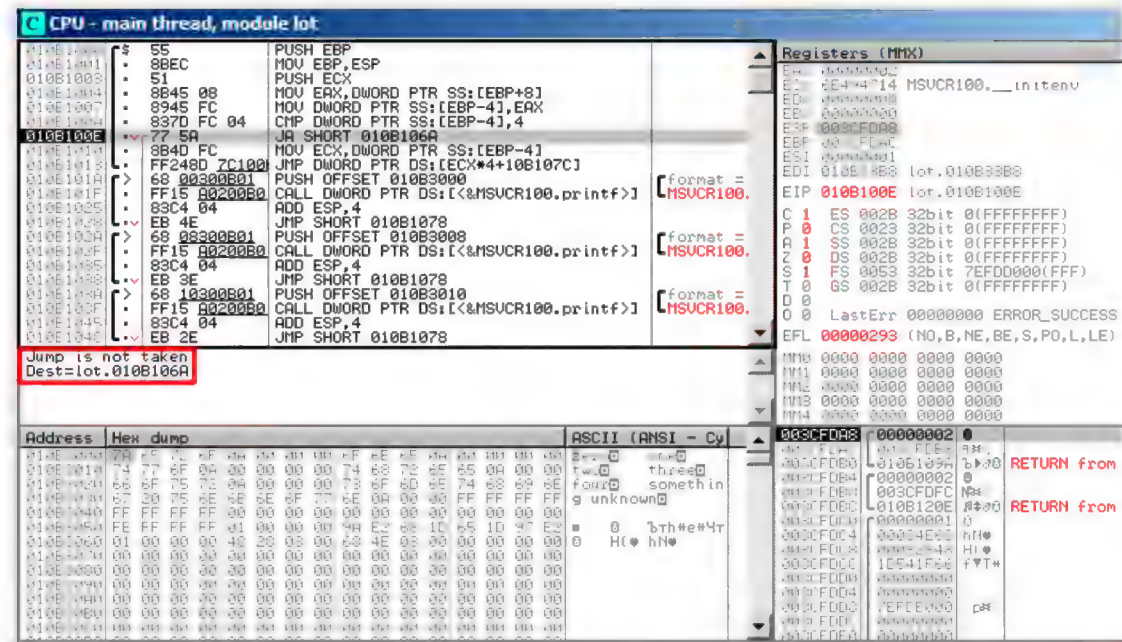


図 1.51: OllyDbg: 2は4より大きいか : ジャンプは実行されない

ここで、ジャンプテーブルを見ることができます。

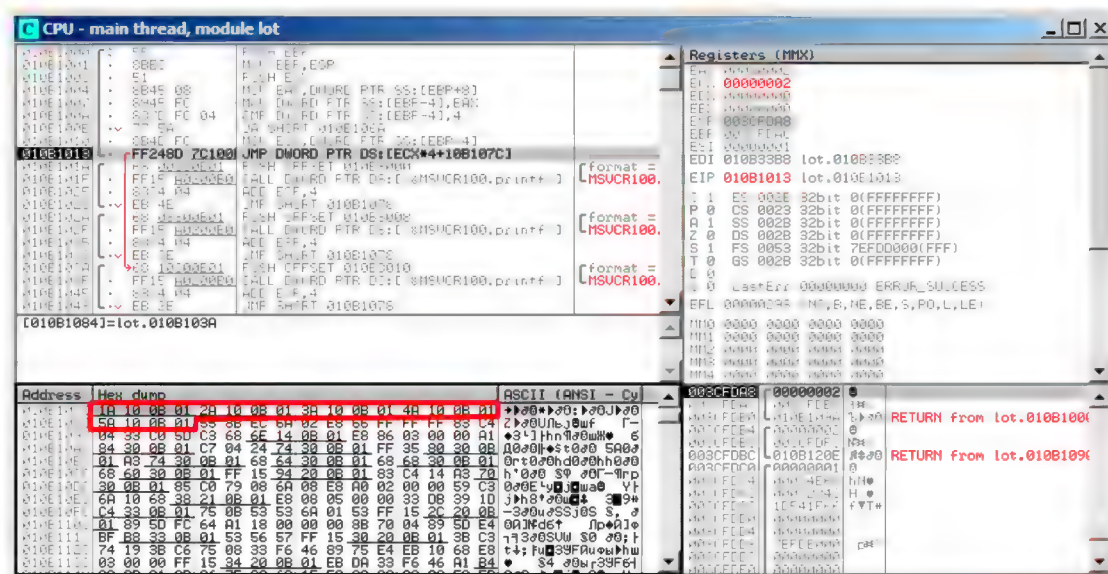


図 1.52: OllyDbg: ジャンプテーブルを用いて行先のアドレスを計算する

ここで、「Follow in Dump」→「Address constant」をチェックします。そして、データウィンドウに `jumpable` が見えます。5つの32ビット値があります。⁹⁴ ECXは2になりました。したがって、テーブルの3番目の要素（2として⁹⁵索引付けできます）が使用されます。「Follow in Dump」→「Memory address」をクリックすることができ、OllyDbgはJMP命令で指示された要素を表示します。それは `0x010B103A` です。

⁹⁴ これらはまた要修正?? on page ??であるため、OllyDbgで下線が引かれています。後でそれらに戻ってくるつもりです

⁹⁵ インデックスについては以下を参照: ?? on page ??

ジャンプの後、0x010B103A にいます。「two」を表示するコードが実行されます。

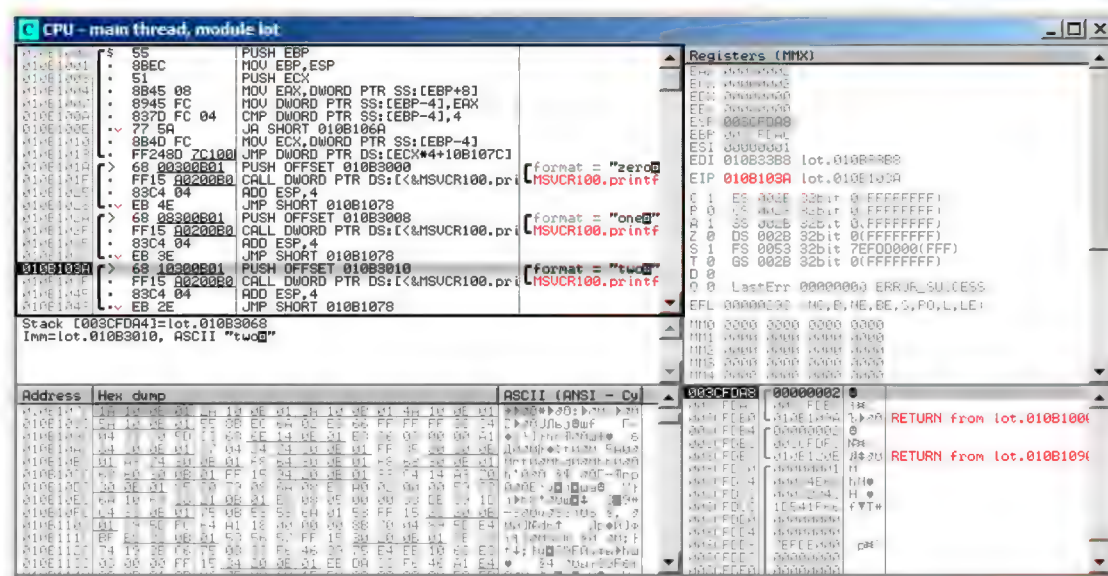


図 1.53: OllyDbg: 今や case: ラベルにいます

非最適化 GCC

GCC 4.4.1が生成するものを見てみましょう：

Listing 1.152: GCC 4.4.1

```
public f
f
proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0 = dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 18h
cmp     [ebp+arg_0], 4
ja      short loc_8048444
mov     eax, [ebp+arg_0]
shl     eax, 2
mov     eax, ds:off_804855C[eax]
jmp     eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
mov     [esp+18h+var_18], offset aZero ; "zero"
call    _puts
jmp     short locret_8048450
```



```

loc_804840C: ; DATA XREF: .rodata:08048560
             mov     [esp+18h+var_18], offset aOne ; "one"
             call    _puts
             jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
             mov     [esp+18h+var_18], offset aTwo ; "two"
             call    _puts
             jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
             mov     [esp+18h+var_18], offset aThree ; "three"
             call    _puts
             jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
             mov     [esp+18h+var_18], offset aFour ; "four"
             call    _puts
             jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
             mov     [esp+18h+var_18], offset aSomethingUnkno ; "something unknown"
             call    _puts

locret_8048450: ; CODE XREF: f+26
               ; f+34...
             leave
             retn
f             endp

off_804855C dd offset loc_80483FE ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436

```

引数 `arg_0` は2ビット左にシフトすることで4倍されます（これは4倍の乗算とほぼ同じです）。(1.18.2 on page 267) `off_804855C` 配列からラベルのアドレスを取り出し、`EAX` に格納してから、`JMP EAX` が実際のジャンプを行います。

ARM: 最適化 Keil 6/2013 (ARMモード)

Listing 1.153: 最適化 Keil 6/2013 (ARMモード)

```

00000174          f2
00000174 05 00 50 E3      CMP     R0, #5          ; switch 5 cases
00000178 00 F1 8F 30      ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA      B       default_case ; jumtable 00000178
                default case

00000180
00000180          loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA      B       zero_case ; jumtable 00000178 case 0

```

```

00000184
00000184          loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA      B      one_case      ; jumtable 00000178 case 1

00000188
00000188          loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA      B      two_case      ; jumtable 00000178 case 2

0000018C
0000018C          loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA      B      three_case     ; jumtable 00000178 case 3

00000190
00000190          loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA      B      four_case     ; jumtable 00000178 case 4

00000194
00000194          zero_case ; CODE XREF: f2+4
00000194          ; f2:loc_180
00000194 EC 00 8F E2      ADR      R0, aZero      ; jumtable 00000178 case 0
00000198 06 00 00 EA      B      loc_1B8

0000019C
0000019C          one_case ; CODE XREF: f2+4
0000019C          ; f2:loc_184
0000019C EC 00 8F E2      ADR      R0, aOne      ; jumtable 00000178 case 1
000001A0 04 00 00 EA      B      loc_1B8

000001A4
000001A4          two_case ; CODE XREF: f2+4
000001A4          ; f2:loc_188
000001A4 01 0C 8F E2      ADR      R0, aTwo      ; jumtable 00000178 case 2
000001A8 02 00 00 EA      B      loc_1B8

000001AC
000001AC          three_case ; CODE XREF: f2+4
000001AC          ; f2:loc_18C
000001AC 01 0C 8F E2      ADR      R0, aThree     ; jumtable 00000178 case 3
000001B0 00 00 00 EA      B      loc_1B8

000001B4
000001B4          four_case ; CODE XREF: f2+4
000001B4          ; f2:loc_190
000001B4 01 0C 8F E2      ADR      R0, aFour      ; jumtable 00000178 case 4
000001B8
000001B8          loc_1B8 ; CODE XREF: f2+24
000001B8          ; f2+2C
000001B8 66 18 00 EA      B      __2printf

000001BC
000001BC          default_case ; CODE XREF: f2+4
000001BC          ; f2+8

```


000001BC D4 00 8F E2	ADR	R0, aSomethingUnkno ; jumptable 00000178
default case 000001C0 FC FF FF EA	B	loc_1B8

このコードでは、すべての命令の固定サイズが4バイトのARMモード機能を使用しています。

a の最大値は4で、それ以上の値を指定すると、「*something unknown*\n」文字列が出力されることに注意しましょう。

最初の CMP R0, #5 命令は、 a の入力値を5と比較します。

⁹⁶ 次の ADDCC PC, PC, R0, LSL #2 命令は、 $R0 < 5$ (CC=Carry clear / Less than) の場合にのみ実行されます。したがって、ADDCC がトリガしない場合 ($R0 \geq 5$ の場合)、default_case ラベルにジャンプします。

しかし $R0 < 5$ と ADDCC がトリガされた場合、次のことが起こります：

R0 の値には4が掛けられます。実際、命令のサフィックスの LSL #2 は「2ビット左シフト」の略です。しかし、セクション「シフト」の (1.18.2 on page 266) で後で見ると、2ビット左シフトは4を乗算するのと同じです。

次に、 $R0 * 4$ を **PC!** の現在の値に追加し、下にある B (Branch) 命令の1つにジャンプします。

ADDCC 命令の実行時に、**PC!** の値は ADDCC 命令が置かれているアドレス (0x178) よりも8バイト先 (0x180) であり、言い換えれば2命令先にあります。

これはARMプロセッサのパイプラインがどのように動作するかを示しています。ADDCC が実行されると、現時点でプロセッサは次の命令の後に命令を処理し始めているので、**PC!** がそこを指しているのはそのためです。これは覚えておく必要があります。

$a = 0$ の場合、**PC!** の値に加算され、**PC!** の実際の値は **PC!** (8バイト先) に書き込まれ、loc_180 というラベルへのジャンプが起こります。これは、ADDCC 命令の先の8バイト先です。

$a = 1$ の場合、**PC!** には $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 12 = 0x184$ が書き込まれます。loc_184 というラベルが付いたアドレスです。

1を a に加えるごとに、結果の**PC!**は4ずつ増加します。

4はARMモードの命令長であり、各 B 命令の長さ4でそれらは5つあります。

これらの5つの B 命令のそれぞれは、制御を switch() にプログラムされたものにさらに渡します。

対応する文字列のポインタローディングが発生します。

ARM: 最適化 Keil 6/2013 (Thumbモード)

Listing 1.154: 最適化 Keil 6/2013 (Thumbモード)

000000F6		EXPORT f2
000000F6	f2	
000000F6 10 B5		PUSH {R4, LR}

⁹⁶ADD—addition

```

000000F8 03 00      MOVS    R3, R0
000000FA 06 F0 69 F8    BL      __ARM_common_switch8_thumb ; switch 6
      cases

000000FE 05          DCB 5
000000FF 04 06 08 0A 0C 10    DCB 4, 6, 8, 0xA, 0xC, 0x10 ; jump table for
      switch_statement
00000105 00          ALIGN 2
00000106
00000106          zero_case ; CODE XREF: f2+4
00000106 8D A0      ADR      R0, aZero ; jumtable 000000FA case 0
00000108 06 E0      B        loc_118

0000010A
0000010A          one_case ; CODE XREF: f2+4
0000010A 8E A0      ADR      R0, aOne ; jumtable 000000FA case 1
0000010C 04 E0      B        loc_118

0000010E
0000010E          two_case ; CODE XREF: f2+4
0000010E 8F A0      ADR      R0, aTwo ; jumtable 000000FA case 2
00000110 02 E0      B        loc_118

00000112
00000112          three_case ; CODE XREF: f2+4
00000112 90 A0      ADR      R0, aThree ; jumtable 000000FA case 3
00000114 00 E0      B        loc_118

00000116
00000116          four_case ; CODE XREF: f2+4
00000116 91 A0      ADR      R0, aFour ; jumtable 000000FA case 4
00000118
00000118          loc_118 ; CODE XREF: f2+12
00000118          ; f2+16
00000118 06 F0 6A F8    BL      __2printf
0000011C 10 BD      POP      {R4,PC}

0000011E
0000011E          default_case ; CODE XREF: f2+4
0000011E 82 A0      ADR      R0, aSomethingUnkno ; jumtable
      000000FA default case
00000120 FA E7      B        loc_118

000061D0          EXPORT __ARM_common_switch8_thumb
000061D0          __ARM_common_switch8_thumb ; CODE XREF:
      example6_f2+4
000061D0 78 47      BX      PC

000061D2 00 00          ALIGN 4
000061D2          ; End of function __ARM_common_switch8_thumb
000061D2
000061D4          32 __ARM_common_switch8_thumb ; CODE XREF:
      __ARM_common_switch8_thumb
000061D4 01 C0 5E E5    LDRB    R12, [LR,#-1]

```

```

000061D8 0C 00 53 E1      CMP      R3, R12
000061DC 0C 30 DE 27      LDRCSB   R3, [LR,R12]
000061E0 03 30 DE 37      LDRCCB   R3, [LR,R3]
000061E4 83 C0 8E E0      ADD      R12, LR, R3,LSL#1
000061E8 1C FF 2F E1      BX       R12
000061E8          ; End of function __32_ARM_common_switch8_thumb

```

ThumbモードとThumb-2モードのすべての命令が同じサイズであることを確認することはできません。これらのモードでは、x86の場合と同様に、命令の長さが可変であるといえます。

したがって、そこにあるケースの数（デフォルトケースを含まない）に関する情報と、対応するケースでコントロールを渡す必要があるラベルを持つそれぞれのオフセットが含まれている特別なテーブルが追加されています。

`__ARM_common_switch8_thumb` という名前のテーブルとパスコントロールを扱うために特別な関数がここにあります。BX PC で始まり、その機能はプロセッサをARMモードに切り替えることです。次に、テーブル処理の機能が表示されます。

今ここで説明するにはあまりにも進んでいるので、省略しましょう。

関数がLRレジスタをテーブルへのポインタとして使用することは興味深いことです。

実際、この関数を呼び出した後、LRにはテーブルが始まる `BL __ARM_common_switch8_thumb` 命令の後のアドレスが入ります。

また、コードを再利用するために別の関数として生成されるので、コンパイラはすべてのswitch() 文に対して同じコードを生成しないことにも注意してください。

IDA はそれをサービス関数とテーブルとして認識し、`jumptable 000000FA case 0` のようなラベルのコメントを追加します。

MIPS

Listing 1.155: 最適化 GCC 4.4.5 (IDA)

```

f:
    lui      $gp, (__gnu_local_gp >> 16)
; 入力値が5未満の場合はloc_24にジャンプ
    sltiu    $v0, $a0, 5
    bnez     $v0, loc_24
    la       $gp, (__gnu_local_gp & 0xFFFF) ; 分岐遅延スロット
; 入力値は5以上
; "something unknown" を表示し、終了
    lui      $a0, ($LC5 >> 16) # "something unknown"
    lw       $t9, (puts & 0xFFFF)($gp)
    or       $at, $zero ; NOP
    jr       $t9
    la       $a0, ($LC5 & 0xFFFF) # "something unknown" ; 分岐遅延
    スロット

loc_24:                                # CODE XREF: f+8
; ジャンプテーブルのアドレスをロードする
; LAは疑似命令で、実際はLUIとADDIUのペアです
    la       $v0, off_120

```

```

; 入力値に4をかける
    sll    $a0, 2
; 掛け算した値とジャンプテーブルのアドレスを足しあわせる
    addu   $a0, $v0, $a0
; ジャンプテーブルから要素をロードする
    lw     $v0, 0($a0)
    or     $at, $zero ; NOP
; ジャンプテーブルで得たアドレスにジャンプする
    jr     $v0
    or     $at, $zero ; 分岐遅延スロット、NOP

sub_44:                                # DATA XREF: .rodata:0000012C
; "three" を表示して終了
    lui    $a0, ($LC3 >> 16) # "three"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC3 & 0xFFFF) # "three" ; 分岐遅延スロット

sub_58:                                # DATA XREF: .rodata:00000130
; "four" を表示して終了
    lui    $a0, ($LC4 >> 16) # "four"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC4 & 0xFFFF) # "four" ; 分岐遅延スロット

sub_6C:                                # DATA XREF: .rodata:off_120
; "zero" を表示して終了
    lui    $a0, ($LC0 >> 16) # "zero"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC0 & 0xFFFF) # "zero" ; 分岐遅延スロット

sub_80:                                # DATA XREF: .rodata:00000124
; "one" を表示して終了
    lui    $a0, ($LC1 >> 16) # "one"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC1 & 0xFFFF) # "one" ; 分岐遅延スロット

sub_94:                                # DATA XREF: .rodata:00000128
; "two" を表示して終了
    lui    $a0, ($LC2 >> 16) # "two"
    lw     $t9, (puts & 0xFFFF)($gp)
    or     $at, $zero ; NOP
    jr     $t9
    la     $a0, ($LC2 & 0xFFFF) # "two" ; 分岐遅延スロット

; おそらく .rodata セクションに配置される
off_120:    .word sub_6C

```

```
.word sub_80
.word sub_94
.word sub_44
.word sub_58
```

私たちの新しい命令は SLTIU です（「Set on Less Than Immediate Unsigned」）。

SLTU と同じですが、「I」は「immediate」を表します。つまり、命令自体に数値を指定する必要があります。

BNEZ は「Branch if Not Equal to Zero」です。

コードは他のISAに似ています。SLL（「Shift Word Left Logical」）は4を掛けます。

結局のところ、MIPSは32ビットCPUなので、*jump table* のすべてのアドレスは32ビットのものです。

結論

switch() の大まかなスケルトン：

Listing 1.156: x86

```
MOV REG, input
CMP REG, 4 ; caseの最大数
JA default
SHL REG, 2 ; テーブルで要素を見つける。x64では3ビットシフト
MOV REG, jump_table[REG]
JMP REG

case1:
    ; 何かする
    JMP exit
case2:
    ; 何かする
    JMP exit
case3:
    ; 何かする
    JMP exit
case4:
    ; 何かする
    JMP exit
case5:
    ; 何かする
    JMP exit

default:
    ...

exit:
    ....
```

```

jump_table dd case1
           dd case2
           dd case3
           dd case4
           dd case5

```

ジャンプテーブルのアドレスへのジャンプはこの命令で用いて実装されるでしょう：JMP jump_table[REG*4] もしくはx64では JMP jump_table[REG*8]。

jump_table は単にポインタの配列で、後で説明します：[1.20.5 on page 346](#)

第1.15.3節あるブロックに複数の **case** 文があるとき

よく用いられる構成があります：単一ブロックにいくつか *case* ステートメントがあります：

```

#include <stdio.h>

void f(int a)
{
    switch (a)
    {
        case 1:
        case 2:
        case 7:
        case 10:
            printf ("1, 2, 7, 10\n");
            break;
        case 3:
        case 4:
        case 5:
        case 6:
            printf ("3, 4, 5\n");
            break;
        case 8:
        case 9:
        case 20:
        case 21:
            printf ("8, 9, 21\n");
            break;
        case 22:
            printf ("22\n");
            break;
        default:
            printf ("default\n");
            break;
    };
};

int main()
{
    f(4);
};

```

可能性のあるケースごとにブロックを生成するのは無駄です。通常は、各ブロックに何らかのディスパッチャーを加えたものを生成します。

MSVC

Listing 1.157: 最適化 MSVC 2010

```

1 $SG2798 DB      '1, 2, 7, 10', 0aH, 00H
2 $SG2800 DB      '3, 4, 5', 0aH, 00H
3 $SG2802 DB      '8, 9, 21', 0aH, 00H
4 $SG2804 DB      '22', 0aH, 00H
5 $SG2806 DB      'default', 0aH, 00H
6
7 _a$ = 8
8 _f PROC
9     mov     eax, DWORD PTR _a$[esp-4]
10    dec     eax
11    cmp     eax, 21
12    ja      SHORT $LN1@f
13    movzx   eax, BYTE PTR $LN10@f[eax]
14    jmp     DWORD PTR $LN11@f[eax*4]
15 $LN5@f:
16    mov     DWORD PTR _a$[esp-4], OFFSET $SG2798 ; '1, 2, 7, 10'
17    jmp     DWORD PTR __imp__printf
18 $LN4@f:
19    mov     DWORD PTR _a$[esp-4], OFFSET $SG2800 ; '3, 4, 5'
20    jmp     DWORD PTR __imp__printf
21 $LN3@f:
22    mov     DWORD PTR _a$[esp-4], OFFSET $SG2802 ; '8, 9, 21'
23    jmp     DWORD PTR __imp__printf
24 $LN2@f:
25    mov     DWORD PTR _a$[esp-4], OFFSET $SG2804 ; '22'
26    jmp     DWORD PTR __imp__printf
27 $LN1@f:
28    mov     DWORD PTR _a$[esp-4], OFFSET $SG2806 ; 'default'
29    jmp     DWORD PTR __imp__printf
30    npad    2 ; $LN11@fテーブルを16バイト境界にアラインメントする
31 $LN11@f:
32    DD      $LN5@f ; '1, 2, 7, 10' を表示
33    DD      $LN4@f ; '3, 4, 5' を表示
34    DD      $LN3@f ; '8, 9, 21' を表示
35    DD      $LN2@f ; '22' を表示
36    DD      $LN1@f ; 'default' を表示
37 $LN10@f:
38    DB      0 ; a=1
39    DB      0 ; a=2
40    DB      1 ; a=3
41    DB      1 ; a=4
42    DB      1 ; a=5
43    DB      1 ; a=6
44    DB      0 ; a=7
45    DB      2 ; a=8
46    DB      2 ; a=9

```

```

47      DB      0 ; a=10
48      DB      4 ; a=11
49      DB      4 ; a=12
50      DB      4 ; a=13
51      DB      4 ; a=14
52      DB      4 ; a=15
53      DB      4 ; a=16
54      DB      4 ; a=17
55      DB      4 ; a=18
56      DB      4 ; a=19
57      DB      2 ; a=20
58      DB      2 ; a=21
59      DB      3 ; a=22
60 _f      ENDP

```

最初のテーブル (\$LN10@f) はインデックステーブルで、2番目のテーブル (\$LN11@f) はブロックへのポインタの配列です。

まず、入力値がインデックステーブルのインデックスとして使用されます (13行目)。

表の値の短い凡例は次のとおりです。0は最初の case ブロックです (値1,2,7,10の場合)。1は2番目の値 (値3,4,5) です。2は3番目の値 (値8,9,21) です。3は4番目の値 (値22) です。4はデフォルトブロック用です。

コードポインタの2番目のテーブルのインデックスを取得し、それにジャンプします (14行目)。

注目すべき点は、入力値0の場合がないことです。

そのため、10行目の DEC 命令が表示され、 $a = 1$ にテーブル要素を割り当てる必要がないため、 $a = 1$ でテーブルが開始されます。

これはよく用いられるパターンです。

それでなぜこれが経済的なのでしょうか？以前はブロックポインタで構成された1つのテーブルだけで、それを作ることができないのはなぜですか？([1.15.2 on page 210](#)) その理由は、インデックステーブルの要素が8ビットで、よりコンパクトなためです。

GCC

GCCはすでに述べた方法で ([1.15.2 on page 210](#))、ポインタのテーブルを1つだけ使用して仕事をしています。

ARM64: 最適化 GCC 4.9.1

入力値が0の場合にトリガされるコードはないので、GCCはジャンプテーブルをよりコンパクトにしようとし、入力値として1から開始します。

ARM64用のGCC 4.9.1は、より巧妙なトリックを使用します。すべてのオフセットを8ビットのバイトとしてエンコードできます。

すべてのARM64命令のサイズが4バイトであることを思い出してみましょう。

GCCは、私の小さな例のすべてのオフセットがお互いに非常に近いという事実を利用して、ジャンプテーブルは1バイトで構成されています。

Listing 1.158: 最適化 GCC 4.9.1 ARM64

```

f14:
; 入力値はw0にある
    sub    w0, w0, #1
    cmp    w0, 21
; 以下の場合に分岐 (unsigned):
    bls    .L9
.L2:
; "default" を表示
    adrp    x0, .LC4
    add     x0, x0, :lo12:.LC4
    b       puts
.L9:
; X1にジャンプテーブルのアドレスをロードする:
    adrp    x1, .L4
    add     x1, x1, :lo12:.L4
; w0=input_value-1
; テーブルからバイトをロード:
    ldrb    w0, [x1,w0,uxtw]
; Lrtxラベルのアドレスをロードする
    adr     x1, .Lrtx4
; テーブルの要素に4をかける (2ビット左シフトすることで)。Lrtxのアドレスに足す (または引く)
    add     x0, x1, w0, sxtb #2
; 計算したアドレスにジャンプする
    br     x0
; このラベルはコード (text) セグメントを指す
.Lrtx4:
    .section      .rodata
; ".section" ステートメントの後にあるものは読み取り専用のセグメント (rodata) に配置されるdata
.L4:
    .byte      (.L3 - .Lrtx4) / 4      ; case 1
    .byte      (.L3 - .Lrtx4) / 4      ; case 2
    .byte      (.L5 - .Lrtx4) / 4      ; case 3
    .byte      (.L5 - .Lrtx4) / 4      ; case 4
    .byte      (.L5 - .Lrtx4) / 4      ; case 5
    .byte      (.L5 - .Lrtx4) / 4      ; case 6
    .byte      (.L3 - .Lrtx4) / 4      ; case 7
    .byte      (.L6 - .Lrtx4) / 4      ; case 8
    .byte      (.L6 - .Lrtx4) / 4      ; case 9
    .byte      (.L3 - .Lrtx4) / 4      ; case 10
    .byte      (.L2 - .Lrtx4) / 4      ; case 11
    .byte      (.L2 - .Lrtx4) / 4      ; case 12
    .byte      (.L2 - .Lrtx4) / 4      ; case 13
    .byte      (.L2 - .Lrtx4) / 4      ; case 14
    .byte      (.L2 - .Lrtx4) / 4      ; case 15
    .byte      (.L2 - .Lrtx4) / 4      ; case 16
    .byte      (.L2 - .Lrtx4) / 4      ; case 17
    .byte      (.L2 - .Lrtx4) / 4      ; case 18
    .byte      (.L2 - .Lrtx4) / 4      ; case 19
    .byte      (.L6 - .Lrtx4) / 4      ; case 20
    .byte      (.L6 - .Lrtx4) / 4      ; case 21

```

```

        .byte    (.L7 - .Lrtx4) / 4      ; case 22
        .text
; ".text" ステートメントの後にあるものはコードセグメント (text) に配置される
.L7:
; "22" を表示
        adrp     x0, .LC3
        add      x0, x0, :lo12:.LC3
        b        puts
.L6:
; "8, 9, 21" を表示
        adrp     x0, .LC2
        add      x0, x0, :lo12:.LC2
        b        puts
.L5:
; "3, 4, 5" を表示
        adrp     x0, .LC1
        add      x0, x0, :lo12:.LC1
        b        puts
.L3:
; "1, 2, 7, 10" を表示
        adrp     x0, .LC0
        add      x0, x0, :lo12:.LC0
        b        puts
.LC0:
        .string  "1, 2, 7, 10"
.LC1:
        .string  "3, 4, 5"
.LC2:
        .string  "8, 9, 21"
.LC3:
        .string  "22"
.LC4:
        .string  "default"

```

この例をオブジェクトファイルにコンパイルし、[IDA](#) で開きましょう。ここにジャンプテーブルがあります：

Listing 1.159: jumptable in IDA

.rodata:0000000000000064	AREA .rodata, DATA, READONLY
.rodata:0000000000000064	; ORG 0x64
.rodata:0000000000000064 \$d	DCB 9 ; case 1
.rodata:0000000000000065	DCB 9 ; case 2
.rodata:0000000000000066	DCB 6 ; case 3
.rodata:0000000000000067	DCB 6 ; case 4
.rodata:0000000000000068	DCB 6 ; case 5
.rodata:0000000000000069	DCB 6 ; case 6
.rodata:000000000000006A	DCB 9 ; case 7
.rodata:000000000000006B	DCB 3 ; case 8
.rodata:000000000000006C	DCB 3 ; case 9
.rodata:000000000000006D	DCB 9 ; case 10
.rodata:000000000000006E	DCB 0xF7 ; case 11
.rodata:000000000000006F	DCB 0xF7 ; case 12
.rodata:0000000000000070	DCB 0xF7 ; case 13

```
.rodata:0000000000000071      DCB 0xF7      ; case 14
.rodata:0000000000000072      DCB 0xF7      ; case 15
.rodata:0000000000000073      DCB 0xF7      ; case 16
.rodata:0000000000000074      DCB 0xF7      ; case 17
.rodata:0000000000000075      DCB 0xF7      ; case 18
.rodata:0000000000000076      DCB 0xF7      ; case 19
.rodata:0000000000000077      DCB      3      ; case 20
.rodata:0000000000000078      DCB      3      ; case 21
.rodata:0000000000000079      DCB      0      ; case 22
.rodata:000000000000007B ; .rodata ends
```

したがって、1の場合、9は4で乗算され、Lrtx4 ラベルのアドレスに追加されます。

22の場合、0には4が掛けられ、結果は0になります。

Lrtx4 ラベルの直後に L7 ラベルがあります。このラベルでは、「22」を出力するコードを見つけることができます。

コードセグメントにはジャンプテーブルはありません。別の.rodataセクションに割り当てられています（コードセクションに配置する特別な必要はありません）。

負のバイト（0xF7）もあり、「default」文字列（.L2）を出力するコードにジャンプするために使用されます。

第1.15.4節フォールスルー

switch() 演算子の別のポピュラーな使い方は「フォールスルー」です。単純なサンプルがあります。⁹⁷:

```
1 bool is_whitespace(char c) {
2     switch (c) {
3         case ' ': // fallthrough
4         case '\t': // fallthrough
5         case '\r': // fallthrough
6         case '\n':
7             return true;
8         default: // not whitespace
9             return false;
10    }
11 }
```

やや難しいものをLinuxカーネル⁹⁸:

```
1 char nco1, nco2;
2
3 void f(int if_freq_khz)
4 {
5
6     switch (if_freq_khz) {
```

⁹⁷https://github.com/azonalon/prgraas/blob/master/progllib/lecture_examples/is_whitespace.cからコピーペースト

⁹⁸<https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c>からコピーペースト

```

7         default:
8             printf("IF=%d KHz is not supported, 3250 assumed\n",
9                 if_freq_khz);
10             /* fallthrough */
11             case 3250: /* 3.25Mhz */
12                 nco1 = 0x34;
13                 nco2 = 0x00;
14                 break;
15             case 3500: /* 3.50Mhz */
16                 nco1 = 0x38;
17                 nco2 = 0x00;
18                 break;
19             case 4000: /* 4.00Mhz */
20                 nco1 = 0x40;
21                 nco2 = 0x00;
22                 break;
23             case 5000: /* 5.00Mhz */
24                 nco1 = 0x50;
25                 nco2 = 0x00;
26                 break;
27             case 5380: /* 5.38Mhz */
28                 nco1 = 0x56;
29                 nco2 = 0x14;
30                 break;
31     };

```

Listing 1.160: Optimizing GCC 5.4.0 x86

```

1  .LC0:
2      .string "IF=%d KHz is not supported, 3250 assumed\n"
3  f:
4      sub    esp, 12
5      mov    eax, DWORD PTR [esp+16]
6      cmp    eax, 4000
7      je     .L3
8      jg     .L4
9      cmp    eax, 3250
10     je     .L5
11     cmp    eax, 3500
12     jne    .L2
13     mov    BYTE PTR nco1, 56
14     mov    BYTE PTR nco2, 0
15     add    esp, 12
16     ret
17  .L4:
18     cmp    eax, 5000
19     je     .L7
20     cmp    eax, 5380
21     jne    .L2
22     mov    BYTE PTR nco1, 86
23     mov    BYTE PTR nco2, 20
24     add    esp, 12
25     ret

```

```

26 .L2:
27     sub     esp, 8
28     push    eax
29     push    OFFSET FLAT:.LC0
30     call    printf
31     add     esp, 16
32 .L5:
33     mov     BYTE PTR nco1, 52
34     mov     BYTE PTR nco2, 0
35     add     esp, 12
36     ret
37 .L3:
38     mov     BYTE PTR nco1, 64
39     mov     BYTE PTR nco2, 0
40     add     esp, 12
41     ret
42 .L7:
43     mov     BYTE PTR nco1, 80
44     mov     BYTE PTR nco2, 0
45     add     esp, 12
46     ret

```

関数の入力に3250という数字がある場合、.L5 ラベルを得ることができます。しかし、我々は反対側からこのラベルに行くことができます：printf() 呼び出しと.L5 ラベルの間にはジャンプがないことがわかります。

switch() 文がバグの原因となることが理解できます。break を1つ忘れるとはあなたの switch() 文をフォールスルーに変換し、1つのブロックの代わりにいくつかのブロックが実行されます。

第1.15.5節練習問題

練習問題 #1

コンパイラがより小さなコードを生成することができるように[1.15.2 on page 204](#)のCの例を修正することは可能ですが、まったく同じように動作します。やってみてください。

第1.16節ループ

第1.16.1節単純な例

x86

x86命令セットにはECXというレジスタが値をチェックする特別なLOOP命令になります。そして0でなければ、[デクリメント](#) ECX し、LOOP オペランドのラベルに制御フローを渡します。おそらくこの命令はあまり便利ではなく、自動的にそれを発行する最新のコンパイラはありません。したがって、コードのどこかでこの命令を見ると、これは手作業で書かれたアセンブリコードである可能性が高いです。

C/C++ のループでは通常 for()、while() または do/while() 文を使用して構成されます。

for() を使ってみましょう。

この文はループの初期化を定義し（ループカウンタを初期値にセット）、ループ条件（がリミットより大きいのか？）が各イテレーション（**インクリメント/デクリメント**）で実行され、ループボディも当然実行されます。

```
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

生成されたコードは4つの部分で構成されています。

簡単な例から始めましょう：

```
#include <stdio.h>

void printing_function(int i)
{
    printf ("f(%d)\n", i);
};

int main()
{
    int i;

    for (i=2; i<10; i++)
        printing_function(i);

    return 0;
};
```

結果 (MSVC 2010):

Listing 1.161: MSVC 2010

```
_i$ = -4
_main PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     DWORD PTR _i$[ebp], 2    ; ループ初期化
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp] ; 各イテレーションの後にくる場所
    add     eax, 1                    ; (i) に1を加える
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 10   ; 各イテレーションの前にこの条件がチェックされる
    jge     SHORT $LN1@main          ; (i) が10以上の場合、ループが終了する
    mov     ecx, DWORD PTR _i$[ebp] ; ループボディ：printing_function(i) を呼び出す
    push    ecx
    call    _printing_function
    add     esp, 4
```

```

    jmp     SHORT $LN2@main      ; ループ開始にジャンプ
$LN1@main:                      ; ループエンド
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP

```

我々が見るように、特別なものはありません。

GCC 4.4.1はほぼ同じコードを出力しますが、微妙な違いが1つあります：

Listing 1.162: GCC 4.4.1

```

main                proc near
var_20              = dword ptr -20h
var_4               = dword ptr -4

    push     ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 20h
    mov     [esp+20h+var_4], 2 ; (i) 初期化
    jmp     short loc_8048476

loc_8048465:
    mov     eax, [esp+20h+var_4]
    mov     [esp+20h+var_20], eax
    call    printing_function
    add     [esp+20h+var_4], 1 ; (i) インクリメント

loc_8048476:
    cmp     [esp+20h+var_4], 9
    jle     short loc_8048465 ; i<=9なら、ループを継続
    mov     eax, 0
    leave
    retn

main                endp

```

最適化を有効にして (/Ox) 取得した内容を見てみましょう。

Listing 1.163: 最適化 MSVC

```

_main     PROC
    push    esi
    mov     esi, 2
$LL3@main:
    push    esi
    call    _printing_function
    inc     esi
    add     esp, 4
    cmp     esi, 10 ; 0000000aH
    jl     SHORT $LL3@main

```

```

    xor    eax, eax
    pop    esi
    ret    0
_main    ENDP

```

ここで起こるのは、*i* 変数のスペースがローカルスタックにはもう割り当てられず、ESI のための個別のレジスタを使用するということです。これは、ローカル変数があまりないような小さな関数で可能です。

とても重要なことは、*f()* 関数が ESI の値を変更してはならないことです。私たちのコンパイラは確かにそうしています。コンパイラが *f()* で ESI レジスタを使用することを決定した場合、その値は関数のプロローグに保存され、関数のエピローグで復元されなければなりません。リストのようになります。関数の開始と終了での PUSH ESI/POP ESI に注意してください。

最適化を最大にしてGCC 4.4.1を試してみましょう (-O3 オプション)：

Listing 1.164: 最適化 GCC 4.4.1

```

main                proc near
var_10              = dword ptr -10h

    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 10h
    mov     [esp+10h+var_10], 2
    call    printing_function
    mov     [esp+10h+var_10], 3
    call    printing_function
    mov     [esp+10h+var_10], 4
    call    printing_function
    mov     [esp+10h+var_10], 5
    call    printing_function
    mov     [esp+10h+var_10], 6
    call    printing_function
    mov     [esp+10h+var_10], 7
    call    printing_function
    mov     [esp+10h+var_10], 8
    call    printing_function
    mov     [esp+10h+var_10], 9
    call    printing_function
    xor     eax, eax
    leave
    retn
main                endp

```

えっ、GCCは単に私たちのループを巻き戻してしまいました。

[Loop unwinding](#) は反復回数が多くなく、すべてのループサポート命令を削除することで実行時間を短縮できる場合に利点があります。逆の場合では、明らかにコードが大きくなります。

大規模な関数は大量のキャッシュフットプリント⁹⁹を必要とする可能性があるため、大きなアンロールループは現代では推奨されません。

OK、*i* 変数の最大値を100に増やして、もう一度試してみましょう。GCCの結果は以下の通り：

Listing 1.165: GCC

```

main                public main
                   proc near
var_20              = dword ptr -20h

                   push    ebp
                   mov     ebp, esp
                   and     esp, 0FFFFFFF0h
                   push    ebx
                   mov     ebx, 2    ; i=2
                   sub     esp, 1Ch

; ラベルloc_80484D0 (ループボディの開始場所) を16バイト境界でアラインメント
                   nop

loc_80484D0:
; (i) を第一引数としてprinting_function() に渡す
                   mov     [esp+20h+var_20], ebx
                   add     ebx, 1    ; i++
                   call    printing_function
                   cmp     ebx, 64h  ; i==100か?
                   jnz     short loc_80484D0 ; そうでなければ継続
                   add     esp, 1Ch
                   xor     eax, eax  ; 0をリターン
                   pop     ebx
                   mov     esp, ebp
                   pop     ebp
                   retn
main                endp

```

これは、EBXレジスタが *i* 変数に割り当てられていることを除いて、最適化ありのMSVC 2010 (/Ox) と非常によく似ています。

GCCはこのレジスタが *f()* 関数の内部で変更されないことをわかっています。もしそうであれば、*main()* 関数のように関数プロローグに保存され、エピローグで復元されます。

⁹⁹非常によい記事: [Ulrich Drepper, *What Every Programmer Should Know About Memory*, (2007)]¹⁰⁰ インテルのループアンローリングに関するその他の推奨事項はこちら: [Intel® 64 and IA-32 Architectures Optimization Reference Manual, (2014)3.4.1.7]

x86: OllyDbg

/Ox と /Ob0 オプションを使用してMSVC 2010のサンプルをコンパイルし、OllyDbg にロードしてみましょう。

OllyDbg は単純なループを検出し、便宜上、角カッコで表示してくれます。

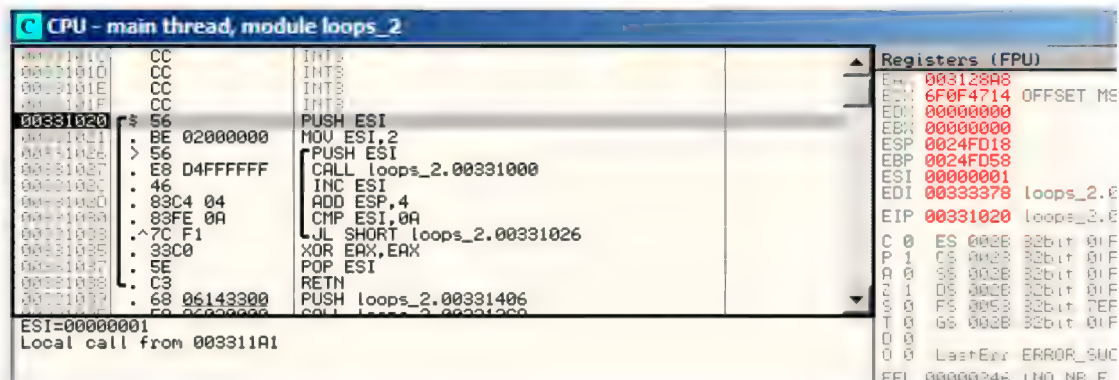


図 1.54: OllyDbg: main() 開始

By tracing (F8 — ステップオーバー) we see ESI **incrementing**. Here, for instance, $ESI = i = 6$:

トレースすることにより (F8。ステップオーバ)、ESIが増加することがわかります。ここで、例えば、 $ESI = i = 6$:

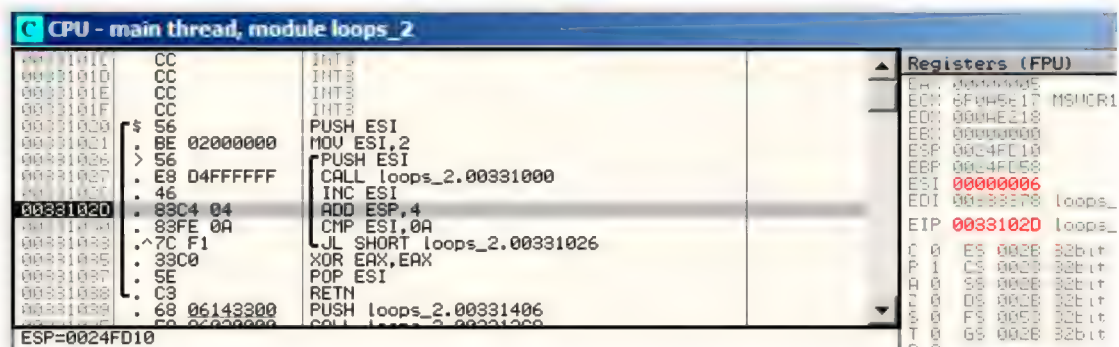
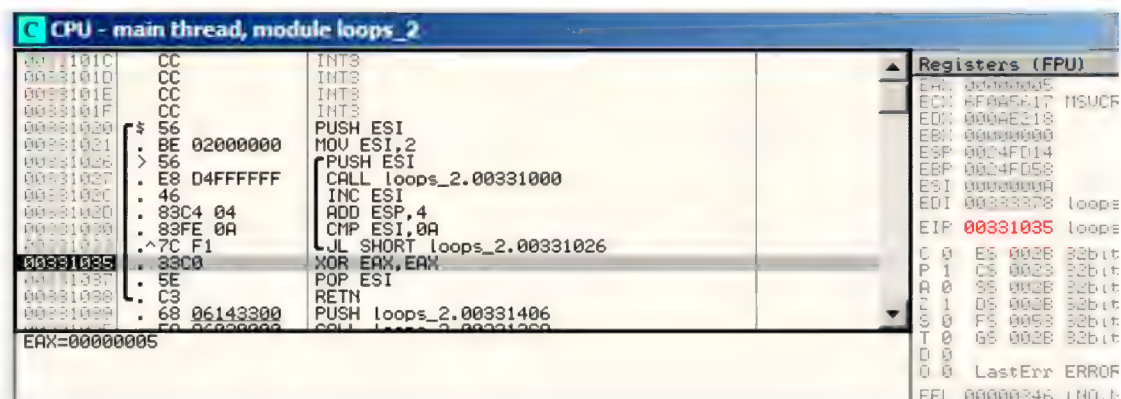


図 1.55: OllyDbg: ループボディが $i = 6$ で実行

9は最後のループ値です。そのため、JL は**インクリメント**後に実行されず、関数は終了します。

図 1.56: OllyDbg: $ESI = 10$ 、ループ終了

x86: tracer

見てきたように、デバッガで手動でトレースするのはあまり便利ではありません。それがトレーサを試みる理由です。

コンパイルされたサンプルを [IDA](#) で開き、命令 `PUSH ESI` (`f()` へ1つ引数を渡す) のアドレスを見つけます。この場合は `0x401026` で、トレーサを実行してみます。

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX はアドレスにブレークポイントを設定するだけで、[tracer](#) はレジスタの状態を出力します。

tracer.log では、このようになります。

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x000a328c EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
```

```
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

ESI レジスタの値が2から9に変化する様子を見えています。

[tracer](#) はそれ以外にも、関数内のすべてのアドレスのレジスタ値を収集できます。これを *trace* といいます。すべての命令がトレースされ、興味深いレジスタ値がすべて記録されます。

次に、コメントを追加する [IDA.idc](#) スクリプトが生成されます。したがって、[IDA](#) では、`main()` 関数のアドレスは `0x00401020` であり、次のように実行されます。

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF は、関数にブレークポイントを設定します。

その結果、`loops_2.exe.idc` および `loops_2.exe_clear.idc` スクリプトが取得されます。

loops_2.exe.idc を IDA にロードすると次のようになります。

```
.text:00401020
.text:00401020 ; ===== SUBROUTINE =====
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main      proc near          ; CODE XREF: __tmainCRTStartup+11D.jp
.text:00401020          = dword ptr  4
.text:00401020 argv       = dword ptr  8
.text:00401020 envp       = dword ptr 0Ch
.text:00401020          push     esi           ; ESI=1
.text:00401021          mov      esi, 2
.text:00401026 loc_401026:          ; CODE XREF: _main+134j
.text:00401026          push     esi           ; ESI=2..9
.text:00401027          call     sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102c          inc      esi           ; ESI=2..9
.text:0040102d          add      esp, 4           ; ESP=0x38fcbc
.text:00401030          cmp      esi, 0Ah         ; ESI=3..0xa
.text:00401033          jl       short loc_401026 ; SF=false,true OF=false
.text:00401035          xor      eax, eax
.text:00401037          pop      esi
.text:00401038          retn                    ; EAX=0
.text:00401038 _main      endp
```

図 1.57: .idc-script を IDA でロードした

ESI はループ本体の開始時には2から9、インクリメント後は3から0xA (10) になります。main() が EAX 0で終了していることもわかります。

tracer はまた、各命令が何回実行されたかに関する情報とレジスタ値を含む loops_2.exe.txt も生成します。

Listing 1.166: loops_2.exe.txt

```
0x401020 (.text+0x20), e=      1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=      1 [MOV ESI, 2]
0x401026 (.text+0x26), e=      8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=      8 [CALL 8D1000h] tracing nested maximum ↗
    ↳ level (1) reached, skipping this CALL 8D1000h=0x8d1000
0x40102c (.text+0x2c), e=      8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=      8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=      8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=      8 [JL 8D1026h] SF=false,true OF=false
0x401035 (.text+0x35), e=      1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=      1 [POP ESI]
0x401038 (.text+0x38), e=      1 [RETN] EAX=0
```

ここではgrepを使うことができます。

ARM

非最適化 Keil 6/2013 (ARMモード)

```

main
    STMFD    SP!, {R4,LR}
    MOV      R4, #2
    B        loc_368
loc_35C      ; CODE XREF: main+1C
    MOV      R0, R4
    BL       printing_function
    ADD      R4, R4, #1

loc_368      ; CODE XREF: main+8
    CMP      R4, #0xA
    BLT      loc_35C
    MOV      R0, #0
    LDMFD    SP!, {R4,PC}

```

ループカウンタ i は R4 レジスタに格納されます。MOV R4, #2 命令は i をちょうど初期化します。MOV R0, R4、および BL printing_function 命令は、f() 関数の引数を準備する最初の命令と2番目の関数を呼び出すループの本体を構成します。ADD R4, R4, #1 命令は、各繰り返しで i 変数に1を加算するだけです。CMP R4, #0xA は i と 0xA (10) を比較します。次の命令 BLT (*Branch Less Than*) は、 i が10未満の場合にジャンプします。それ以外の場合は、R0 に0が書き込まれます (関数が0を返すため)。そして関数の実行が終了します。

最適化 Keil 6/2013 (Thumbモード)

```

_main
    PUSH     {R4,LR}
    MOVS     R4, #2

loc_132      ; CODE XREF: _main+E
    MOVS     R0, R4
    BL       printing_function
    ADDS     R4, R4, #1
    CMP      R4, #0xA
    BLT      loc_132
    MOVS     R0, #0
    POP      {R4,PC}

```

実質的に同じです。

最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

```

_main
    PUSH     {R4,R7,LR}
    MOVW     R4, #0x1124 ; "%d\n"
    MOVS     R1, #2
    MOVT.W   R4, #0
    ADD      R7, SP, #4

```

```

ADD      R4, PC
MOV      R0, R4
BLX      _printf
MOV      R0, R4
MOVS     R1, #3
BLX      _printf
MOV      R0, R4
MOVS     R1, #4
BLX      _printf
MOV      R0, R4
MOVS     R1, #5
BLX      _printf
MOV      R0, R4
MOVS     R1, #6
BLX      _printf
MOV      R0, R4
MOVS     R1, #7
BLX      _printf
MOV      R0, R4
MOVS     R1, #8
BLX      _printf
MOV      R0, R4
MOVS     R1, #9
BLX      _printf
MOVS     R0, #0
POP      {R4,R7,PC}

```

実際、これは私の `f()` 関数にありました：

```

void printing_function(int i)
{
    printf ("%d\n", i);
};

```

だから、LLVMはループを展開しただけでなく、私の非常に単純な関数 `f()` をインライン化し、呼び出すのではなく本体を8回挿入しました。

これは、関数が（私の例のように）とても簡単で、（ここのように）あまり呼び出されないときに可能です。

ARM64: 最適化 GCC 4.9.1

Listing 1.167: 最適化 GCC 4.9.1

```

printing_function:
; printf() の第二引数を準備:
    mov     w1, w0
; "f(%d)\n" 文字列のアドレスをロード
    adrp    x0, .LC0
    add     x0, x0, :lo12:.LC0
; 分岐とリターンの命令ではなく単なる分岐の命令
    b       printf

```

```

main:
; ローカルスタックにFPとLRを保存
    stp    x29, x30, [sp, -32]!
; スタックフレームを設定
    add    x29, sp, 0
; X19レジスタの内容をローカルスタックに保存
    str    x19, [sp, 16]
; W19レジスタをカウンタとして利用します
; W19に初期値として2を設定します
    mov    w19, 2
.L3:
; printing_function() の最初の引数を準備します:
    mov    w0, w19
; カウンタレジスタをインクリメント
    add    w19, w19, 1
; W0はインクリメントする前のカウンタの値を保持しています
    bl     printing_function
; 終了?
    cmp    w19, 10
; 終了でなければ、ループボディの開始にジャンプ
    bne    .L3
; 0をリターン
    mov    w0, 0
; X19レジスタの内容をリストア
    ldr    x19, [sp, 16]
; FPとLRの値をリストア
    ldp    x29, x30, [sp], 32
    ret
.LC0:
    .string "f(%d)\n"

```

ARM64: 非最適化 GCC 4.9.1

Listing 1.168: 非最適化 GCC 4.9.1 -fno-inline

```

.LC0:
    .string "f(%d)\n"
printing_function:
; ローカルスタックにFPとLRを保存します
    stp    x29, x30, [sp, -32]!
; スタックフレームを設定します
    add    x29, sp, 0
; W0レジスタの内容を保存します
    str    w0, [x29, 28]
; "f(%d)\n" 文字列のアドレスをロードします
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; ローカルスタックから入力値をリロードしてW1レジスタに書き込みます
    ldr    w1, [x29, 28]
; printf() を呼び出します
    bl     printf
; FPとLRの値をリストアします

```



```

        ldp    x29, x30, [sp], 32
        ret
main:
; ローカルスタックにFPとLRを保存します
        stp    x29, x30, [sp, -32]!
; スタックフレームを設定します
        add    x29, sp, 0
; カウンタを初期化
        mov    w0, 2
; ローカルスタック上にアロケートされたところに保存
        str    w0, [x29, 28]
; ループボディをスキップし、ループ条件チェック命令にジャンプします
        b      .L3
.L4:
; w0にカウンタ値をロードします
; printing_function() への第一引数になります
        ldr    w0, [x29, 28]
; printing_function() を呼び出します
        bl     printing_function
; カウンタ値をインクリメント
        ldr    w0, [x29, 28]
        add    w0, w0, 1
        str    w0, [x29, 28]
.L3:
; ループ条件のチェック
; カウンタ値のロード
        ldr    w0, [x29, 28]
; 9かどうか
        cmp    w0, 9
; 9以下か？そうならループボディの開始にジャンプ
; そうでなければ何もしない
        ble    .L4
; 0をリターン
        mov    w0, 0
; FPとLR値をリストア
        ldp    x29, x30, [sp], 32
        ret

```

MIPS

Listing 1.169: GCC 4.4.5 非最適化 (IDA)

```

main:
; IDAはローカルスタック内の変数名を認識しません
; 手動で名前をつけます
i      = -0x10
saved_FP = -8
saved_RA = -4

; 関数プロローグ:
        addiu   $sp, -0x28
        sw      $ra, 0x28+saved_RA($sp)

```

```

        sw      $fp, 0x28+saved_FP($sp)
        move    $fp, $sp
; 2でカウンタを初期化し、この値をローカルスタックに格納する
        li      $v0, 2
        sw      $v0, 0x28+i($fp)
; 疑似命令、実際には"BEQ $ZERO, $ZERO, loc_9C"
        b       loc_9C
        or      $at, $zero ; 分岐遅延スロット, NOP

loc_80:                                     # CODE XREF: main+48
; ローカルスタックからカウンタの値を読み込み、print_function() を呼び出します。
        lw      $a0, 0x28+i($fp)
        jal     printing_function
        or      $at, $zero ; 分岐遅延スロット, NOP
; カウンタをロードし、インクリメントし、カウンタを保存します
        lw      $v0, 0x28+i($fp)
        or      $at, $zero ; NOP
        addiu   $v0, 1
        sw      $v0, 0x28+i($fp)

loc_9C:                                     # CODE XREF: main+18
; カウンタが10かチェック
        lw      $v0, 0x28+i($fp)
        or      $at, $zero ; NOP
        slti    $v0, 0xA
; 10未満の場合、loc_80(ループボディ開始) にジャンプ:
        bnez    $v0, loc_80
        or      $at, $zero ; 分岐遅延スロット, NOP
; 終了して0をリターン:
        move    $v0, $zero
; 関数エピローグ:
        move    $sp, $fp
        lw      $ra, 0x28+saved_RA($sp)
        lw      $fp, 0x28+saved_FP($sp)
        addiu   $sp, 0x28
        jr      $ra
        or      $at, $zero ; 分岐遅延スロット, NOP

```

新しい命令は B です。実際には疑似命令 (BEQ) です。

もう一つ

生成されたコードでは、*i* を初期化した後、*i* の条件が最初にチェックされ、ループ本体が実行された後にのみ、ループの本体が実行されないことがわかります。それは正しいです。

ループの条件が最初に満たされない場合、ループの本体は実行されてはならないからです。これは次の場合に可能です：

```

for (i=0; i<total_entries_to_process; i++)
    loop_body;

```

total_entries_to_process が0の場合、ループの本体はまったく実行されてはなりません。

これは、実行前に条件がチェックされている理由です。

しかし、最適化されたコンパイラは、ここで説明した状況が不可能であることが確かな場合（例えばKeil、Xcode（LLVM）、MSVCなどのコンパイラを最適化モードで使用する場合など）、条件チェックとループ本体をスワップできます。

第1.16.2節メモリブロックコピールーチン

実世界のメモリコピールーチンは、反復ごとに4バイトまたは8バイトをコピーし、SIMD¹⁰¹、ベクトル化などを使用します。

しかし、話を簡単にするために、この例は可能な限り簡単にしています。

```
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};
```

簡単な実装

Listing 1.170: GCC 4.9 x64 optimized for size (-Os)

```
my_memcpy:
; RDI = コピー先アドレス
; RSI = コピー元アドレス
; RDX = ブロックのサイズ

; カウンタ (i) を0で初期化
xor     eax, eax
.L2:
; 全バイトをコピーしたら終了する
cmp     rax, rdx
je      .L5
; RSI+iのバイトをロードする:
mov     cl, BYTE PTR [rsi+rax]
; バイトをRDI+iに保存する
mov     BYTE PTR [rdi+rax], cl
inc     rax ; i++
jmp     .L2
.L5:
ret
```

Listing 1.171: GCC 4.9 ARM64 optimized for size (-Os)

```
my_memcpy:
; X0 = コピー先アドレス
; X1 = コピー元アドレス
```

¹⁰¹Single Instruction, Multiple Data

```

; X2 = ブロックのサイズ
; カウンタ (i) を0で初期化
    mov    x3, 0
.L2:
; 全バイトをコピーしたら終了する
    cmp    x3, x2
    beq    .L5
; X1+iのバイトをロードする:
    ldrb    w4, [x1,x3]
; バイトをX0+iに保存する
    strb    w4, [x0,x3]
    add    x3, x3, 1 ; i++
    b      .L2
.L5:
    ret

```

Listing 1.172: 最適化 Keil 6/2013 (Thumbモード)

```

my_memcpy PROC
; R0 = コピー先アドレス
; R1 = コピー元アドレス
; R2 = ブロックのサイズ

    PUSH    {r4,lr}
; カウンタ (i) を0で初期化
    MOVS    r3,#0
; 関数の終わりで条件がチェックされ、ジャンプする
    B       |L0.12|
|L0.6|
; R1+iのバイトをロードする:
    LDRB    r4,[r1,r3]
; バイトをR0+iに保存する
    STRB    r4,[r0,r3]
; i++
    ADDS    r3,r3,#1
|L0.12|
; i<size?
    CMP     r3,r2
; もしそうならループ開始にジャンプする
    BCC     |L0.6|
    POP     {r4,pc}
    ENDP

```

ARM in ARM mode

ARMモードのKeilは、条件付き接尾辞を最大限に活用しています。

Listing 1.173: 最適化 Keil 6/2013 (ARMモード)

```

my_memcpy PROC
; R0 = コピー先アドレス
; R1 = コピー元アドレス

```

```

; R2 = ブロックのサイズ
; カウンタ (i) を0で初期化
    MOV     r3,#0
|L0.4|
; 全バイトをコピーしたか
    CMP     r3,r2
; 以下のブロックは未満の条件を満たす場合のみ実行される
; R2<R3 または i<sizeなら
; R1+iのバイトをロードする:
    LDRBCC  r12,[r1,r3]
; バイトをR0+iに保存する
    STRBCC  r12,[r0,r3]
; i++
    ADDCC   r3,r3,#1
; conditional block の最後の命令
; i<sizeならループの開始にジャンプ
; そうでなければ何もしない (i>=sizeなら)
    BCC     |L0.4|
; リターン
    BX      lr
    ENDP

```

そのため、2ではなく1つの分岐命令しか存在しません。

MIPS

Listing 1.174: GCC 4.4.5 optimized for size (-Os) (IDA)

```

my_memcpy:
; ループへのジャンプのチェック部分
    b       loc_14
; カウンタ (i) を0で初期化
; 必ず $v0に割り当たる:
    move    $v0, $zero ; 分岐遅延スロット

loc_8:
; $t0 から $v1に符号なしのバイトとしてロードされる:
    lbu     $v1, 0($t0)
; カウンタ (i) をインクリメント:
    addiu   $v0, 1
; $a3にバイトを保存
    sb      $v1, 0($a3)

loc_14:
; $v0のカウンタ (i) が依然として小さければ、第三引数 ($a2の"cnt") をチェック
    sltu    $v1, $v0, $a2
; ソースブロックのバイトのアドレスを整える
    addu    $t0, $a1, $v0
; $t0 = $a1+$v0 = src+i
; カウンタが"cnt" 未満の場合はループボディにジャンプ
    bnez    $v1, loc_8
; コピー先ブロック ($a3 = $a0+$v0 = dst+i) のバイトのアドレスを整える

```

```

        addu    $a3, $a0, $v0 ; 分岐遅延スロット
; BNEZが実行されなければ終了する
        jr      $ra
        or      $at, $zero ; 分岐遅延スロット, NOP

```

ここでは、2つの新しい命令があります：LBU（「Load Byte Unsigned」）と SB（「Store Byte」）

ARMと同様に、MIPSレジスタはすべて32ビット幅であり、x86などの1バイト幅のレジスタ部分はありません。

したがって、1バイトを処理する場合は、32ビットのレジスタ全体を割り当てる必要があります。

LBU は1バイトをロードし、他のすべてのビット（「Unsigned」）をクリアします。

一方、LB（「Load Byte」）命令は、ロードされたバイトを32ビット値に符号拡張します。

SB は、レジスタの最下位8ビットから1バイトをメモリに書き込むだけです。

ベクトル化

最適化 GCCはこの例でもっと多くのことを行うことができます: [1.27.1 on page 502](#).

第1.16.3節条件チェック

`for()` コンストラクトでは、ループボディの実行前に、条件が最後ではなく、最初からチェックされることに注意してください。しかし、多くの場合、コンパイラはボディの後ろでそれを確認する方が便利です。場合によっては、最初に追加チェックを付け足すこともできます。

例：

```

#include <stdio.h>

void f(int start, int finish)
{
    for (; start<finish; start++)
        printf ("%d\n", start);
};

```

最適化されたGCC 5.4.0 x64では：

```

f:
; check condition (1):
    cmp     edi, esi
    jge     .L9
    push    rbp
    push    rbx
    mov     ebp, esi
    mov     ebx, edi
    sub     rsp, 8

```

```

.L5:      mov     edx, ebx
          xor     eax, eax
          mov     esi, OFFSET FLAT:.LC0 ; "%d\n"
          mov     edi, 1
          add     ebx, 1
          call    __printf_chk
; check condition (2):
          cmp     ebp, ebx
          jne     .L5
          add     rsp, 8
          pop     rbx
          pop     rbp
.L9:      rep ret

```

checkを2つ見てきました。

Hex-Rays (バージョン2.2.0より後) では次のようにデコンパイルします：

```

void __cdecl f(unsigned int start, unsigned int finish)
{
    unsigned int v2; // ebx@2
    __int64 v3; // rdx@3

    if ( (signed int)start < (signed int)finish )
    {
        v2 = start;
        do
        {
            v3 = v2++;
            _printf_chk(1LL, "%d\n", v3);
        }
        while ( finish != v2 );
    }
}

```

この場合、*do/while()* は *for()* で置き換えられ、間違いなく最初のチェックを取り除くことができます。

第1.16.4節結論

2から9のループの大まかなスケルトン：

Listing 1.175: x86

```

mov [counter], 2 ; 初期化
jmp check
body:
; ループボディ
; ここで何かする
; ローカルスタックでカウンタ変数を使用する
add [counter], 1 ; インクリメント

```

```

check:
    cmp [counter], 9
    jle body

```

インクリメント操作は、最適化されていないコードでは3つの命令として表すことができます。

Listing 1.176: x86

```

MOV [counter], 2 ; 初期化
JMP check
body:
    ; ループボディ
    ; ここで何かする
    ; ローカルスタックでカウンタ変数を使用する
    MOV REG, [counter] ; インクリメント
    INC REG
    MOV [counter], REG
check:
    CMP [counter], 9
    JLE body

```

ループの本体が短い場合は、レジスタ全体をカウンタ変数専用にすることができます。

Listing 1.177: x86

```

MOV EBX, 2 ; 初期化
JMP check
body:
    ; ループボディ
    ; ここで何かする
    ; EBXでカウンタを使用しますが、変更してはいけません
    INC EBX ; インクリメント
check:
    CMP EBX, 9
    JLE body

```

ループの一部はコンパイラによって異なる順序で生成されることがあります。

Listing 1.178: x86

```

MOV [counter], 2 ; 初期化
JMP label_check
label_increment:
    ADD [counter], 1 ; インクリメント
label_check:
    CMP [counter], 10
    JGE exit
    ; ループボディ
    ; ここで何かする
    ; ローカルスタックでカウンタ変数を使用する
    JMP label_increment
exit:

```


通常、条件はループ本体の前でチェックされますが、ループ本体の後で条件がチェックされるようにコンパイラが再配置することがあります。

これは、ループの本体が少なくとも1回は実行されるように、最初の反復で条件が常に *true* であるとコンパイラが確信するときに行われます。

Listing 1.179: x86

```
MOV REG, 2 ; 初期化
body:
; ループボディ
; ここで何かする
; REGでカウンタを使用しますが、変更してはいけません
INC REG ; インクリメント
CMP REG, 10
JL body
```

LOOP 命令を使用します。これはまれですが、コンパイラはそれを使用していません。あなたがこれを見る時は、コードは手書きであるというサインです。

Listing 1.180: x86

```
; 10から1へのカウンタ
MOV ECX, 10
body:
; ループボディ
; ここで何かする
; ECXでカウンタを使用しますが、変更してはいけません
LOOP body
```

ARM.

この例では、R4 レジスタはカウンタ変数専用です。

Listing 1.181: ARM

```
MOV R4, 2 ; 初期化
B check
body:
; ループボディ
; ここで何かする
; R4をカウンタに使用しますが、変更してはいけません
ADD R4,R4, #1 ; インクリメント
check:
CMP R4, #10
BLT body
```

第1.16.5節練習問題

- <http://challenges.re/54>
- <http://challenges.re/55>
- <http://challenges.re/56>
- <http://challenges.re/57>

第1.17節文字列に関する加筆

第1.17.1節strlen()

ループについてもう一度話しましょう。多くの場合、strlen() 関数 ¹⁰² は while() 文を使用して実装されます。ここでは、MSVC標準ライブラリでどのように行われるのかを見てみます。

```
int my_strlen (const char * str)
{
    const char *eos = str;

    while( *eos++ );

    return( eos - str - 1 );
}

int main()
{
    // test
    return my_strlen("hello!");
};
```

x86

非最適化 MSVC

コンパイルしてみましょう。

```
_eos$ = -4 ; size = 4
_str$ = 8 ; size = 4
_strlen PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp] ; "str" から文字列へのポインタを配置
    mov     DWORD PTR _eos$[ebp], eax ; " ローカル変数"eos" に配置
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp] ; ECX=eos

    ; ECXのアドレスから8ビットのバイトを取り出し、符号付き32ビットの値としてEDXに配置する
    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add     eax, 1 ; EAXをインクリメント
    mov     DWORD PTR _eos$[ebp], eax ; EAXの値を"eos" に戻す
    test    edx, edx ; EDXはゼロか
    je      SHORT $LN1@strlen_ ; そうなら、ループを終了する
    jmp     SHORT $LN2@strlen_ ; ループを継続する
$LN1@strlen_:
```

¹⁰²文字列中の文字をC言語で数える

```

; ここで二つのポインタの差分を計算する
mov     eax, DWORD PTR _eos$[ebp]
sub     eax, DWORD PTR _str$[ebp]
sub     eax, 1                ; 1を引いて結果をリターンする
mov     esp, ebp
pop     ebp
ret     0
_strlen_ ENDP

```

新しい命令が2つ出てきました、MOVSX と TEST です。

最初の MOVSX は、メモリ内のアドレスからバイトを取り出し、その値を32ビットレジスタに格納します。MOVSX は *MOV with Sign-Extend* の略です。MOVSX はソースバイトが負の場合は1に、正の場合は0に残りのビットを8から31に設定します。

そして、それが理由です。

デフォルトでは、*char* 型はMSVCとGCCで署名されています。一方が *char* で他方が *int* である2つの値がある場合 (*int* も署名されます)、最初の値に-2 (0xFE としてコード化されています) が含まれ、このバイトを *int* コンテナにコピーするだけで 0x000000FE、これは符号付きの *int* ビューのポイントから254ですが、-2ではありません。符号付き *int* の場合、-2は 0xFFFFFFF2E としてコード化されます。したがって、*char* 型の変数から 0xFE を *int* に転送する必要がある場合は、その符号を識別して拡張する必要があります。これが MOVSX の役割です。

コンパイラが EDX に *char* 変数を格納する必要があるかどうかを言うのは難しいですが、8ビットのレジスタ部分 (DL など) を取ることができます。どうやら、コンパイラの [register allocator](#) はそのように機能します。

次に、TEST EDX, EDX が表示されます。TEST 命令の詳細については、ビットフィールドに関するセクション ([1.21 on page 369](#)) を参照してください。ここでは、この命令は EDX の値が0に等しいかどうかをチェックするだけです。

非最適化 GCC

GCC 4.4.1で試してみましょう。

```

strlen      public strlen
            proc near

eos          = dword ptr -4
arg_0       = dword ptr  8

            push    ebp
            mov     ebp, esp
            sub     esp, 10h
            mov     eax, [ebp+arg_0]
            mov     [ebp+eos], eax

loc_80483F0:
            mov     eax, [ebp+eos]
            movzx   eax, byte ptr [eax]

```

```

        test    al, al
        setnz   al
        add     [ebp+eos], 1
        test    al, al
        jnz     short loc_80483F0
        mov     edx, [ebp+eos]
        mov     eax, [ebp+arg_0]
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        sub     eax, 1
        leave
        retn
strlen  endp

```

結果はMSVCとほぼ同じですが、MOVSVX の代わりに MOVZX があります。MOVZX は *MOV with Zero-Extend* の略です。この命令は、8ビットまたは16ビットの値を32ビットレジスタにコピーし、残りのビットを0に設定します。実際、この命令は以下の命令ペアを置き換えることができることでのみ役立ちます：xor eax, eax / mov al, [...]

一方、コンパイラがこのコードを生成できることは明らかです。mov al, byte ptr [eax] / test al, al はほぼ同じですが、EAX レジスタの最上位ビットにランダムノイズが含まれます。しかし、それがコンパイラの欠点だと考えてみましょう。より理解しやすいコードを生成することはできません。厳密に言えば、コンパイラは（人間に）理解できるコードを出力する義務は全くありません。

次の新しい命令は SETNZ です。ここで、AL にゼロが含まれていない場合、test al, al は ZF フラグを0に設定しますが、ZF==0 (NZ はゼロではない) の場合、SETNZ は AL に1を設定します。自然言語で言えば、AL がゼロでなければ、loc_80483F0にジャンプせよとなります。コンパイラは冗長なコードを出力しますが、最適化がオフになっていることを忘れないでください。

最適化 MSVC

さて、最適化をオンにして、MSVC 2012ですべてコンパイルしましょう (/Ox)。

Listing 1.182: 最適化 MSVC 2012 /Ob0

```

_str$ = 8 ; size = 4
_strlen PROC
    mov     edx, DWORD PTR _str$[esp-4] ; 文字列へのポインタをEDXに置く
    mov     eax, edx ; EAXにコピー
$LL2@strlen:
    mov     cl, BYTE PTR [eax] ; CL = *EAX
    inc     eax ; EAX++
    test    cl, cl ; CL==0か
    jne     SHORT $LL2@strlen ; そうでなければ、ループを継続する
    sub     eax, edx ; ポインタの差分を計算する
    dec     eax ; EAXをデクリメントする
    ret     0
_strlen ENDP

```

すべて単純です。言うまでもなく、コンパイラは、ローカル変数が少数である小さな関数でのみ効率を持つレジスタを使用することができます。

INC/DEC はインクリメント/デクリメント命令です。すなわち、変数に1を加算または減算します。

最適化 MSVC + OllyDbg

OllyDbg でこの（最適化された）例を試すことができます。ここに最初のイテレーションがあります：

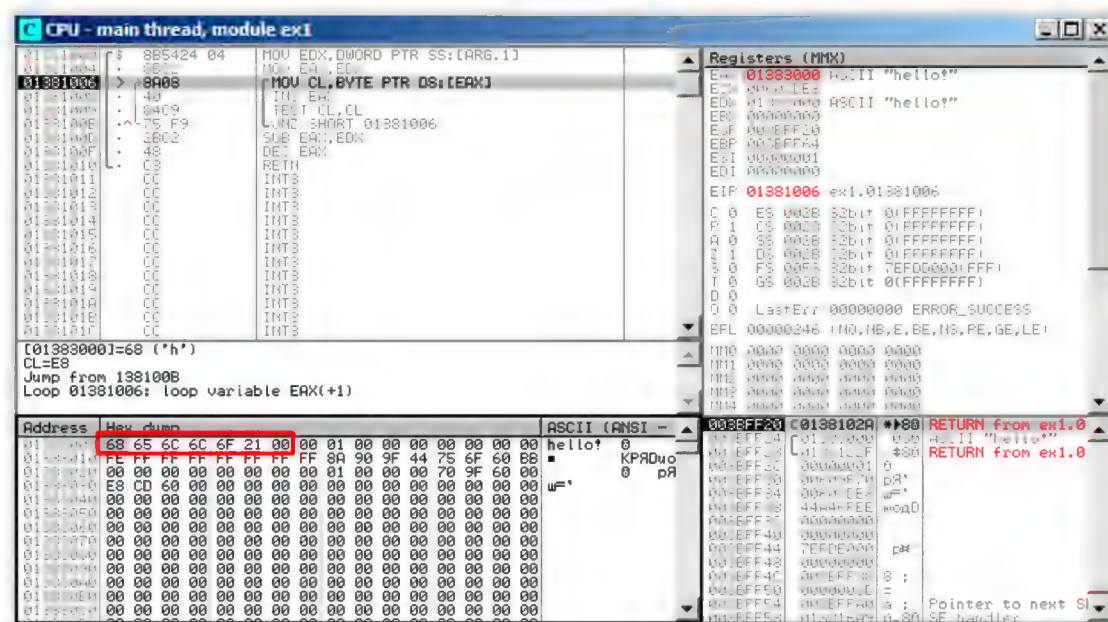


図 1.58: OllyDbg: 最初のイテレーションの開始

OllyDbg がループを見つけ、便宜上、その指示を括弧で囲んだことがわかります。EAX の右ボタンをクリックして、「Follow in Dump」を選択すると、メモリウィンドウが正しい場所にスクロールします。ここではメモリ内に「hello!」という文字列があります。その後少なくとも1つのゼロバイトがあり、次にランダムなごみがあります。

OllyDbg が有効なアドレスを持つレジスタを見ると、それは文字列を指しており、文字列として表示されます。

F8（ステップオーバー）を数回押して、ループの本体の先頭に移動しましょう：

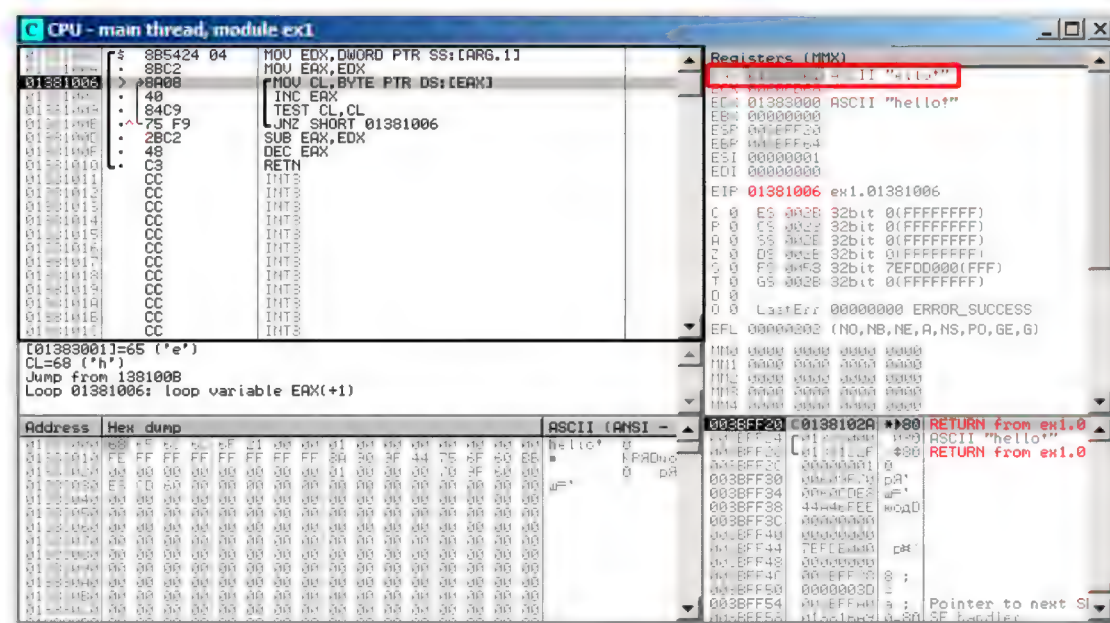


図 1.59: OllyDbg: 2回目のイテレーションの開始

EAX には文字列中の2番目の文字のアドレスが含まれていることがわかります。

我々はループから脱出するためにF8を十分な回数押す必要があります：

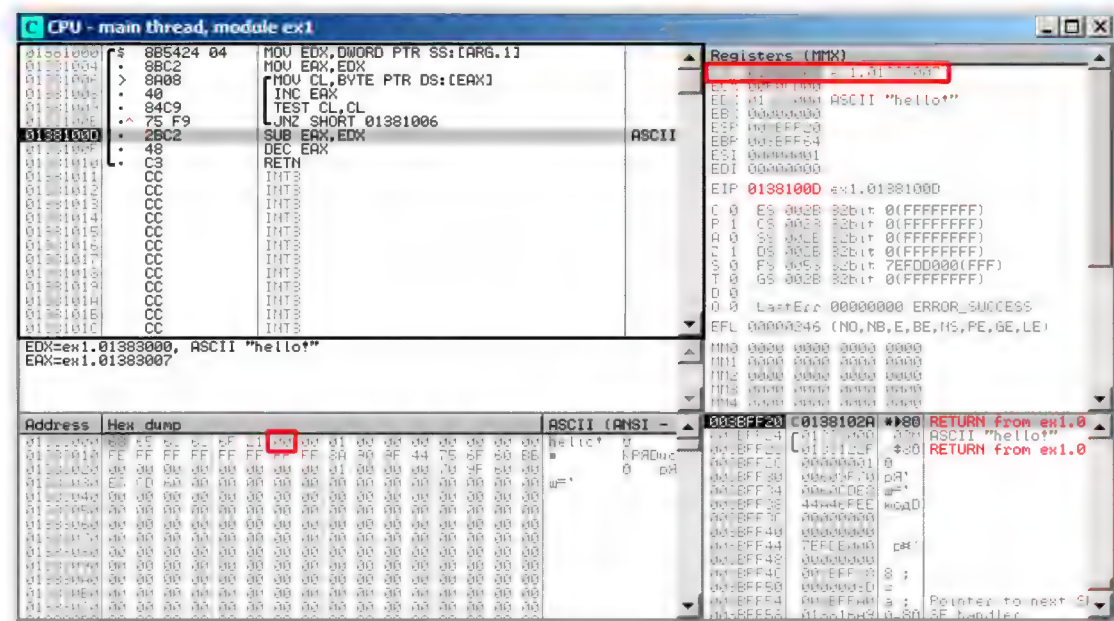


図 1.60: OllyDbg: 計算すべきポインタの差

EAX には文字列の直後に0バイトのアドレスが含まれることがわかりました。一方、EDX は変更されていないので、まだ文字列の先頭を指しています。

これらの2つのアドレスの差がここで計算されています。

SUB 命令が実行されました。

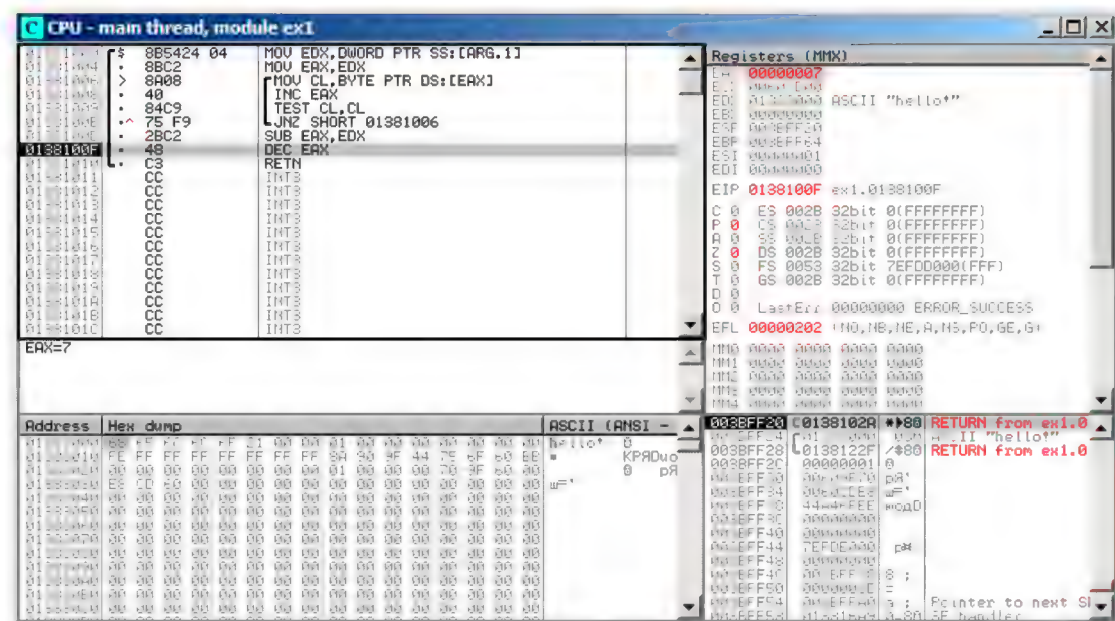


図 1.61: OllyDbg: EAX がデクリメントされる

ポインタの違いは EAX レジスタにあります (値は7)。確かに、「hello!」文字列の長さは6ですが、7にはゼロバイトが含まれています。しかし、strlen() は文字列中のゼロ以外の文字数を返さなければなりません。したがって、デクリメントが実行され、関数が戻ります。

最適化 GCC

最適化をオンにして (-O3 キー) GCC 4.4.1をチェックしましょう。

```

public strlen
proc near

arg_0      = dword ptr 8

push      ebp
mov       ebp, esp
mov       ecx, [ebp+arg_0]
mov       eax, ecx

loc_8048418:
movzx    edx, byte ptr [eax]
add      eax, 1
test     dl, dl
jnz      short loc_8048418
not      ecx

```

```

                                add    eax, ecx
                                pop    ebp
                                retn
strlen                          endp

```

ここで、GCCはMOVZXの存在を除いてMSVCとほぼ同じです。しかし、ここではMOVZXをmov dl, byte ptr [eax]に置き換えることができます。

おそらく、GCCのコードジェネレータがchar変数に割り当てられた32ビットEDXレジスタ全体を記憶していることを覚えているのはおそらく簡単です。そして、最も高いビットにはいつでもノイズがないことを確かめることができます。

その後で、新しい命令を見ます、NOTです。この命令は、オペランドのすべてのビットを反転します。XOR ECX, 0xffffffffh 命令と同義です。NOT と次の ADD はポインタの差を計算し、1を差し引きます。str へのポインタが格納されている開始 ECX では反転され、1が減算されます。

つまり、ループ本体の直後の関数の最後では、これらの操作が実行されます。

```

ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax

```

... これは事実上次のものと同等です：

```

ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax

```

なぜGCCはそれがより良いと判断したのでしょうか。推測するのは難しいですが、おそらく両方の変種は効率の面では同じです。

ARM

32-bit ARM

非最適化 Xcode 4.6.3 (LLVM) (ARMモード)

Listing 1.183: 非最適化 Xcode 4.6.3 (LLVM) (ARMモード)

```

_strlen
eos  = -8
str  = -4

SUB    SP, SP, #8 ; ローカル変数に8バイトを割り当て

```

```

STR    R0, [SP,#8+str]
LDR    R0, [SP,#8+str]
STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
LDR    R0, [SP,#8+eos]
ADD    R1, R0, #1
STR    R1, [SP,#8+eos]
LDRSB  R0, [R0]
CMP    R0, #0
BEQ    loc_2CD4
B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
LDR    R0, [SP,#8+eos]
LDR    R1, [SP,#8+str]
SUB    R0, R0, R1 ; R0=eos-str
SUB    R0, R0, #1 ; R0=R0-1
ADD    SP, SP, #8 ; 割り当てた8バイトを開放
BX     LR

```

最適化されていないLLVMはあまりにも多くのコードを生成しますが、スタック内のローカル変数でこの関数がどのように機能するかを見ることができます。関数には、*eos* と *str* という2つのローカル変数しかありません。IDA によって生成されたこのリストでは、*var_8* と *var_4* の名前を *eos* と *str* に手動で変更しました。

最初の命令は、*str* と *eos* の両方に入力値を保存するだけです。

ループの本体はラベル *loc_2CB8* から始まります。

ループ本体の最初の3つの命令 (LDR、ADD、STR) は、*eos* の値を *R0* にロードします。次に、値が*incremented*され、スタックに格納されている *eos* に保存されます。

次の命令、LDRSB *R0*, [*R0*] (「Load Register Signed Byte」) は、*R0* に格納されたアドレスのメモリからバイトをロードし、32ビットに符号拡張します。¹⁰³ これは、x86の MOVSX 命令に似ています。

コンパイラは *char* 型がC標準に従って署名されているため、このバイトを符号付きとして扱います。このセクションでは、x86との関連で、すでにそれについて書かれています (1.17.1 on page 247)。

ARMの32ビットレジスタのうち8ビットまたは16ビットの部分を、レジスタ全体とは別にx86では使用することは不可能であることに注意してください。

どうやら、x86はその祖先と16ビットの8086、さらには8ビットの8080までの下位互換性の巨大な歴史がありますが、ARMは32ビットのRISCプロセッサとして最初から開発されています。

したがって、ARMでは別々のバイトを処理するために、いずれにしても32ビットレジスタを使用する必要があります。

よって、LDRSB は文字列からバイトを *R0* に1つずつロードします。次の CMP 命令と BEQ 命令は、ロードされたバイトが0かどうかをチェックします。0でなければ、制御はループ本体の先頭に移ります。0の場合、ループは終了します。

¹⁰³Keilコンパイラは、MSVCやGCCのように *char* 型をsigned型として扱います

関数の最後に、*eos* and *str* の差が計算され、1が減算され、結果の値が *R0* を介して返されます。

注意：この関数ではレジスタは保存されませんでした。

これは、ARMの呼び出し規約では、レジスタ *R0-R3* が引数の受け渡しを目的とした「スクラッチレジスタ」であり、呼び出し関数がそれらを使用しなくなるため、関数が終了するときに値を復元する必要はないからです。したがって、それらは私たちが望むものに使用することができます。

ここでは他のレジスタは使用されていないので、スタックに何も保存する必要はありません。

したがって、制御は、単純ジャンプ (BX) によって呼び出し機能に戻され、LRレジスタ内のアドレスに戻ることができます。

最適化 Xcode 4.6.3 (LLVM) (Thumbモード)

Listing 1.184: 最適化 Xcode 4.6.3 (LLVM) (Thumbモード)

```

_strlen
    MOV        R1, R0

loc_2DF6
    LDRB.W     R2, [R1], #1
    CMP        R2, #0
    BNE        loc_2DF6
    MVNS       R0, R0
    ADD        R0, R1
    BX         LR

```

LLVMの最適化が完了すると、*eos* と *str* はスタックにスペースを必要とせず、常にレジスタに格納することができます。

ループ本体の開始前に、*str* は常に *R0* にあり、*eos* は *R1* にあります。

LDRB.W *R2*, [*R1*], #1 命令は、*R1* に格納されたアドレスのメモリから *R2* にバイトをロードし、32ビット値に符号拡張しますが、それだけではありません。命令の最後の #1 は「索引付け後のアドレス指定」を意味します。つまり、バイトがロードされた後に *R1* に1が追加されます。詳しくは、[1.30.2 on page 536](#)を参照してください。

次に、ループの本体に CMP と BNE¹⁰⁴が表示されます。これらの命令は、文字列に0が見つかるまでループし続けます。

MVNS¹⁰⁵ (x86では NOT のようにすべてのビットを反転) および ADD 命令で $eos - str - 1$ が計算されます。実際には、これらの2つの命令は、 $R0 = str + eos$ を計算します。これは、ソースコードのものと事実上同じです。なぜそうであるかについては、以下ですでに説明しました ([1.17.1 on page 254](#))。

どうやら、LLVMはGCCのように、このコードが短く（または高速に）できると結論づけています。

¹⁰⁴(PowerPC, ARM) Branch if Not Equal

¹⁰⁵MoVe Not

最適化 Keil 6/2013 (ARMモード)

Listing 1.185: 最適化 Keil 6/2013 (ARMモード)

```

_strlen
        MOV        R1, R0

loc_2C8
        LDRB       R2, [R1], #1
        CMP        R2, #0
        SUBEQ      R0, R1, R0
        SUBEQ      R0, R0, #1
        BNE        loc_2C8
        BX         LR

```

以前に見たものとほぼ同じですが、*str - eos - 1* の式は関数の終わりではなくループの本体で計算できる点が異なります。-EQ サフィックスは、以前に実行された CMP 内のオペランドが互いに等しい場合にのみ、命令が実行されることを意味します。したがって、R0 が0を含む場合、両方の SUBEQ 命令が実行され、結果は R0 レジスタに置かれます。

ARM64

最適化 GCC (Linaro) 4.9

```

my_strlen:
        mov        x1, x0
        ; X1は一時的なポインタ (eos) で、ツア cursor ツアのようにふるまう
.L58:
        ; X1 から W2にバイトをロードし、X1 (post-index) をインクリメント
        ldrb       w2, [x1], 1
        ; ゼロでなければ分岐: W2と0を比較し、異なれば.L58にジャンプ
        cbnz       w2, .L58
        ; X0の初期ポインタとX1の現在のアドレスとの差分を計算
        sub        x0, x1, x0
        ; 結果の下位32ビットをデクリメント
        sub        w0, w0, #1
        ret

```

アルゴリズムは、[1.17.1 on page 248](#)と同じです。ゼロバイトを見つけて、ポインタの差を計算し、結果を1減らします。この本の著者によっていくつかのコメントが追加されました。

注目すべきは、私たちの例が少し間違っていることです。my_strlen() は32ビット int を返しますが、size_t または別の64ビット型を返す必要があります。

その理由は、理論的には、strlen() は4GBを超える膨大なメモリブロックに対して呼び出すことができるため、64ビットプラットフォームで64ビットの値を返すことができないからです。

私の間違いのために、最後の SUB 命令はレジスタの32ビット部分で動作し、最後から2番目の SUB 命令は64ビットレジスタ全体で動作します（ポインタ間の差を計算します）。

間違いですが、そのような場合にコードがどのように見えるかの例として、そのまま残す方が良いでしょう。

非最適化 GCC (Linaro) 4.9

```
my_strlen:
; 関数プロローグ
    sub    sp, sp, #32
; 最初の引数 (str) が [sp,8] に保存される
    str    x0, [sp,8]
    ldr    x0, [sp,8]
; "str" が "eos" 変数にコピー
    str    x0, [sp,24]
    nop
.L62:
; eos++
    ldr    x0, [sp,24] ; load "eos" to X0
    add    x1, x0, 1   ; X0をインクリメント
    str    x1, [sp,24] ; save X0 to "eos"
; X0のアドレスのメモリからバイトをW0にロード
    ldrb   w0, [x0]
; ゼロか? (WZRは32ビットレジスタで常にゼロを含む)
    cmp    w0, wzr
; ゼロでなければジャンプ (一致しなければ分岐)
    bne    .L62
; ゼロバイトが見つかる。差分を計算
; "eos" をX1にロード
    ldr    x1, [sp,24]
; "str" をX0にロード
    ldr    x0, [sp,8]
; 差分を計算
    sub    x0, x1, x0
; 結果をデクリメント
    sub    w0, w0, #1
; 関数エピローグ
    add    sp, sp, 32
    ret
```

もっと冗長です。変数は、ここではメモリ（ローカルスタック）との間で頻繁に投げられます。同じミスもあります。デクリメント操作は32ビットのレジスタ部分で行われます。

MIPS

Listing 1.186: 最適化 GCC 4.4.5 (IDA)

```
my_strlen:
; "eos" 変数は常に $v1に配置
    move   $v1, $a0
```

```

loc_4:
; "eos" のアドレスのバイトを $a1にロード
        lb      $a1, 0($v1)
        or      $at, $zero ; load delay slot, NOP
; ロードされたバイトがゼロでなければ、loc_4にジャンプ
        bnez    $a1, loc_4
; "eos" をとにかくインクリメント
        addiu   $v1, 1 ; branch delay slot
; ループが終了。"str" 変数を反転させる
        nor     $v0, $zero, $a0
; $v0=-str-1
        jr      $ra
; 戻り値 = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
        addu    $v0, $v1, $v0 ; branch delay slot

```

MIPSには NOT 命令がありませんが、NOR は OR + NOT 演算です。

この操作はデジタルエレクトロニクス¹⁰⁶で広く使用されています。

たとえば、Apolloプログラムで使用されているApolloガイダンス・コンピュータは、5600個のNORゲートを使用して作成されました：[Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)] を参照。しかし、NOR要素はコンピュータプログラミングであまり一般的ではありません。

したがって、NOT演算は NOR DST, \$ZERO, SRC として実装されています。

基本から、符号付き数値のビット反転は、符号の変更と結果からの1の減算と同じであることがわかります。

ですから、ここでは *str* の値をとり、それを $-str - 1$ に変換することはしません。次の加算演算は結果を準備します。

第1.17.2節文字列境界

パラメータがwin32の *GetOpenFileName()* 関数にどのように渡されるかは興味深いことです。それを呼び出すには、許可されたファイル拡張子のリストを設定する必要があります：

```

OPENFILENAME *LPOPENFILENAME;
...
char * filter = "Text files (*.txt)\0*.txt\0MS Word files (*.doc)↵
↵ \0*.doc\0\0";
...
LPOPENFILENAME = (OPENFILENAME *)malloc(sizeof(OPENFILENAME));
...
LPOPENFILENAME->lpstrFilter = filter;
...

if(GetOpenFileName(LPOPENFILENAME))
{
    ...
}

```

¹⁰⁶NORは「汎用ゲート」と呼ばれます

ここでは、`GetOpenFileName()` に文字列のリストが渡されます。解析するのは問題ではありません。ゼロバイトが1つ出現するたびに、これがアイテムになります。ゼロバイトが2バイト出現すれば、リストの最後になります。この文字列を `printf()` に渡すと、最初の項目は単一の文字列として扱われます。

これは文字列か、そうではないのか。これは、複数のゼロ終端されたC文字列を含むバッファであり、全体として格納して処理することができます。

もう一つの例は `strtok()` 関数です。文字列をとり、その真ん中にゼロバイトを書き込みます。したがって、入力文字列をいくつかの種類のバッファに変換します。バッファには、ゼロで終了するC文字列がいくつかあります。

第1.18節算術命令を他の命令に置換する

最適化を追求する際には、ある命令を別の命令に置き換えたり、命令群で置き換えることもできます。たとえば、`ADD` と `SUB` は相互に置き換えることができます：リスト?? の行18を参照

たとえば、`LEA` 命令は、単純な算術計算によく使用されます：?? on page ??

第1.18.1節乗算

加算による乗算

単純な例です。

```
unsigned int f(unsigned int a)
{
    return a*8;
};
```

8倍の乗算は3つの加算命令に置き換えられます。どうやら、MSVCのオプティマイザは、このコードがより高速になると判断したようです。

Listing 1.187: 最適化 MSVC 2010

```
_TEXT    SEGMENT
_a$ = 8      ; size = 4
_f        PROC
    mov     eax, DWORD PTR _a$[esp-4]
    add     eax, eax
    add     eax, eax
    add     eax, eax
    ret     0
_f        ENDP
_TEXT     ENDS
END
```


ビットシフトによる乗算

2のべき乗の数による乗算および除算命令は、多くの場合、シフト命令に置き換えられます。

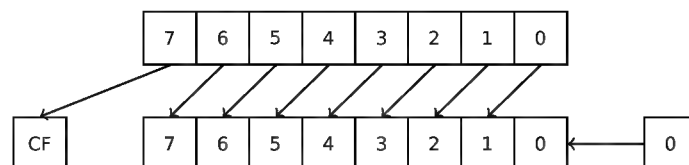
```
unsigned int f(unsigned int a)
{
    return a*4;
};
```

Listing 1.188: 非最適化 MSVC 2010

```
_a$ = 8          ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _a$[ebp]
    shl     eax, 2
    pop     ebp
    ret     0
_f ENDP
```

4の乗算は、数値を左に2ビットシフトし、右に0を2ビット（最後の2ビットとして）を挿入します。それはちょうど3を100倍するのに似ています — 右に2つのゼロを追加するだけです。

これが左シフト命令の仕組みです：



右の追加ビットは常にゼロです。

ARMでの4倍の乗算：

Listing 1.189: 非最適化 Keil 6/2013 (ARMモード)

```
f PROC
    LSL     r0, r0, #2
    BX      lr
    ENDP
```

MIPSでの4倍の乗算：

Listing 1.190: 最適化 GCC 4.4.5 (IDA)

```
jr      $ra
sll     $v0, $a0, 2 ; branch delay slot
```

SLL は「Shift Left Logical」の略です。

シフト、減算、加算を使用した乗算

シフトを使って7や17のような数値を掛けると、乗算演算を取り除くことができます。ここで使用される数学は比較的簡単です。

32-bit

```
#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};
```

x86

Listing 1.191: 最適化 MSVC 2012

```
; a*7
_a$ = 8
_f1 PROC
    mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
    lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
    sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    ret     0
_f1 ENDP

; a*28
_a$ = 8
_f2 PROC
    mov     ecx, DWORD PTR _a$[esp-4]
; ECX=a
    lea     eax, DWORD PTR [ecx*8]
; EAX=ECX*8
    sub     eax, ecx
; EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    shl     eax, 2
```

```

; EAX=EAX<<2=(a*7)*4=a*28
    ret    0
_f2    ENDP

; a*17
_a$ = 8
_f3    PROC
    mov    eax, DWORD PTR _a$[esp-4]
; EAX=a
    shl    eax, 4
; EAX=EAX<<4=EAX*16=a*16
    add    eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a=a*16+a=a*17
    ret    0
_f3    ENDP

```

ARM

ARMモードのKeilは、第2オペランドのシフト修飾子を利用しています。

Listing 1.192: 最適化 Keil 6/2013 (ARMモード)

```

; a*7
||f1|| PROC
    RSB    r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    BX     lr
    ENDP

; a*28
||f2|| PROC
    RSB    r0,r0,r0,LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    LSL    r0,r0,#2
; R0=R0<<2=R0*4=a*7*4=a*28
    BX     lr
    ENDP

; a*17
||f3|| PROC
    ADD    r0,r0,r0,LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
    BX     lr
    ENDP

```

しかし、Thumbモードにはこのような修飾語はありません。また、f2() を最適化することもできません。

Listing 1.193: 最適化 Keil 6/2013 (Thumbモード)

```

; a*7
||f1|| PROC

```

```

        LSLS      r1,r0,#3
; R1=R0<<3=a<<3=a*8
        SUBS      r0,r1,r0
; R0=R1-R0=a*8-a=a*7
        BX        lr
        ENDP

; a*28
||f2|| PROC
        MOVS      r1,#0x1c ; 28
; R1=28
        MULS      r0,r1,r0
; R0=R1*R0=28*a
        BX        lr
        ENDP

; a*17
||f3|| PROC
        LSLS      r1,r0,#4
; R1=R0<<4=R0*16=a*16
        ADDS      r0,r0,r1
; R0=R0+R1=a+a*16=a*17
        BX        lr
        ENDP

```

MIPS

Listing 1.194: 最適化 GCC 4.4.5 (IDA)

```

_f1:
        sll       $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
        jr        $ra
        subu      $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

_f2:
        sll       $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
        sll       $a0, 2
; $a0 = $a0<<2 = $a0*4
        jr        $ra
        subu      $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

_f3:
        sll       $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
        jr        $ra
        addu      $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17

```

64-bit

```
#include <stdint.h>

int64_t f1(int64_t a)
{
    return a*7;
};

int64_t f2(int64_t a)
{
    return a*28;
};

int64_t f3(int64_t a)
{
    return a*17;
};
```

x64

Listing 1.195: 最適化 MSVC 2012

```
; a*7
f1:
    lea    rax, [0+rdi*8]
; RAX=RDI*8=a*8
    sub    rax, rdi
; RAX=RAX-RDI=a*8-a=a*7
    ret

; a*28
f2:
    lea    rax, [0+rdi*4]
; RAX=RDI*4=a*4
    sal    rdi, 5
; RDI=RDI<<5=RDI*32=a*32
    sub    rdi, rax
; RDI=RDI-RAX=a*32-a*4=a*28
    mov    rax, rdi
    ret

; a*17
f3:
    mov    rax, rdi
    sal    rax, 4
; RAX=RAX<<4=a*16
    add    rax, rdi
; RAX=a*16+a=a*17
    ret
```

ARM64

ARM64用のGCC 4.9も、シフト修飾子のおかげで簡潔です。

Listing 1.196: 最適化 GCC (Linaro) 4.9 ARM64

```
; a*7
f1:
    lsl    x1, x0, 3
; X1=X0<<3=X0*8=a*8
    sub    x0, x1, x0
; X0=X1-X0=a*8-a=a*7
    ret

; a*28
f2:
    lsl    x1, x0, 5
; X1=X0<<5=a*32
    sub    x0, x1, x0, lsl 2
; X0=X1-X0<<2=a*32-a<<2=a*32-a*4=a*28
    ret

; a*17
f3:
    add    x0, x0, x0, lsl 4
; X0=X0+X0<<4=a+a*16=a*17
    ret
```

ブースの乗算アルゴリズム

コンピュータが大きくて高価だった時がありました。その中には、Data General NovaのようなCPUでの乗算演算のハードウェアサポートが欠けていたものもありました。そして、乗算が必要なとき、例えば、ブースの乗算アルゴリズムを使用してソフトウェアレベルで乗算を提供することができます。これは、加算演算とシフトのみを使用する乗算アルゴリズムです。

現代の最適化コンパイラが行うことは同じではありませんが、目標（乗算）とリソース（高速化）は同じです。

第1.18.2節除算

ビットシフトによる除算

4で除算する例：

```
unsigned int f(unsigned int a)
{
    return a/4;
};
```

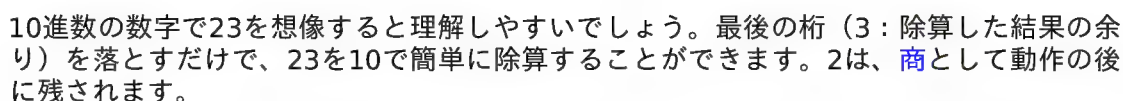
MSVC 2010の結果

```

_a$ = 8      ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    shr     eax, 2
    ret     0
_f ENDP

```

SHR 命令は SHL のように動作しますが、別の方向に動作します。



ARMでの4の除算

```
f PROC
    LSR     r0,r0,#2
    BX      lr
ENDP
```

```
jr      $ra
srl     $v0, $a0, 2 ; branch delay slot
```

第1.18.3節練習問題

- <http://challenges.re/59>

第1.19節フローティングポイントユニット

FPUはメインCPU内のデバイスで、特に浮動小数点数を扱うように設計されています。

これは過去に「コプロセッサ」と呼ばれていましたが、メインCPUとは別の場所にとどまっています。

第1.19.1節IEEE 754

IEEE 754形式の数字は、符号、有効数字（小数部とも呼ばれます）、および指数で構成されます。

第1.19.2節x86

x86のFPUを学ぶ前に、スタックマシンを調べたり、Forth言語の基礎を学ぶことは価値があります。

過去（80486 CPUの前）のコプロセッサは別個のチップであり、いつもマザーボードにプリインストールされていなかったことは興味深いことです。別に購入してインストールすることができました。¹⁰⁷

FPUは80486 DX CPUからCPUに統合されています。

FWAIT 命令は、CPUを待機状態に切り替えるので、FPUが処理を終了するまで待つことができるという事実を思い出させます。

もう一つの基本は、FPU命令オペコードが、いわゆる「エスケープ」オペコード（D8..DF）、すなわちオペコードが別個のコプロセッサに渡されることから始まるという事実です。

FPUは8個の80ビットレジスタを保持できるスタックを持ち、各レジスタはIEEE 754 形式の番号を保持できます。

それらはST(0)..ST(7)です。簡潔には、IDAとOllyDbgはST(0)をSTと表示します。これは一部の教科書とマニュアルでは「スタックトップ」として表されています。

第1.19.3節ARM, MIPS, x86/x64 SIMD

ARMおよびMIPSでは、FPUはスタックではなく、GPRのようにランダムアクセスできるレジスタのセットです。

同じ体系がx86/x64 CPUのSIMD拡張で使用されています。

第1.19.4節C/C++

標準のC/C++ 言語では、float（単精度、32ビット）と¹⁰⁹ double（倍精度、64ビット）の少なくとも2つの浮動小数点型が用意されています。

¹⁰⁷たとえば、John Carmackは、32ビットGPR¹⁰⁸レジスタ（整数部分は16ビット、小数部分は16ビット）に格納されたDoomビデオゲームの固定小数点演算の値を使用しました。DoomはFPUなしの32ビットコンピュータ、つまり80386と80486 SXで動作しました

¹⁰⁹単精度浮動小数点数形式も フロート型のデータを構造体として扱う (1.23.6 on page 455) セクションで扱います

[Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997)246]において、単精度は、浮動小数点値を単一の [32ビット] マシンに入れることができることを意味するワード、倍精度は、2ワード（64ビット）で格納できることを意味します。

GCCはMSVCがサポートしない *long double* 型（拡張精度、80ビット）もサポートしています。

float 型は、32ビット環境では *int* 型と同じビット数が必要ですが、数値表現はまったく異なります。

第1.19.5節簡単な例

この簡単な例を考えてみましょう。

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

x86

MSVC

MSVC 2010でコンパイルしましょう。

Listing 1.200: MSVC 2010: f()

```
CONST    SEGMENT
__real@4010666666666666 DQ 0401066666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _a$[ebp]
; 現在のスタック状態: ST(0) = _a
    fdiv    QWORD PTR __real@40091eb851eb851f
```

```
; 現在のスタック状態: ST(0) = _aを3.14で割った結果
```

```
fld    QWORD PTR _b$[ebp]
```

```
; 現在のスタック状態: ST(0) = _b;
```

```
; ST(1) = _aを3.14で割った結果
```

```
fmul   QWORD PTR __real@4010666666666666
```

```
; 現在のスタック状態:
```

```
; ST(0) = _b * 4.1の結果;
```

```
; ST(1) = _aを3.14で割った結果
```

```
faddp  ST(1), ST(0)
```

```
; 現在のスタック状態: ST(0) = 加算の結果
```

```
pop    ebp
```

```
ret    0
```

```
_f    ENDP
```

FLD はスタックから8バイトを取り出し、その数値を ST(0) レジスタにロードし、内部80ビットフォーマット（拡張精度）に自動的に変換します。

FDIV は、ST(0) の値をアドレス __real@40091eb851eb851f に格納された数値で除算します。値3.14はそこにエンコードされます。アセンブリ構文は浮動小数点数をサポートしていないので、64ビットIEEE 754形式での3.14の16進表現です。

FDIV ST(0) の実行後に商が保持されます。

ちなみに、FDIVP 命令もあります。これは、ST(1) を ST(0) で除算し、これらの値をスタックからポップし、その結果をプッシュします。あなたがForth言語を知っていれば、すぐにこれがスタックマシンであることがわかります。

後続の FLD 命令は、b の値をスタックにプッシュします。

その後、商は ST(1) に置かれ、ST(0) は b の値を持ちます。

次の FMUL 命令は乗算を行います。ST(0) の b は __real@4010666666666666 （そこには4.1が入る）の値で乗算され、結果は ST(0) レジスタに残ります。

最後の FADDP 命令は、スタックの先頭に2つの値を加算し、結果を ST(1) に格納した後、ST(0) の値をポップし、ST(0) のスタックの先頭に結果を残します。

関数はその結果を ST(0) レジスタに戻す必要があるため、FADDP 後の関数エピローグ以外の命令はありません。

MSVC + OllyDbg

2組の32ビットワードは、スタック内で赤色でマークされます。各ペアはIEEE 754形式の倍数で、main() から渡されます。

最初の FLD がスタックからどのように値 (1.2) をロードし、それを ST(0) に入れるかを確認します。

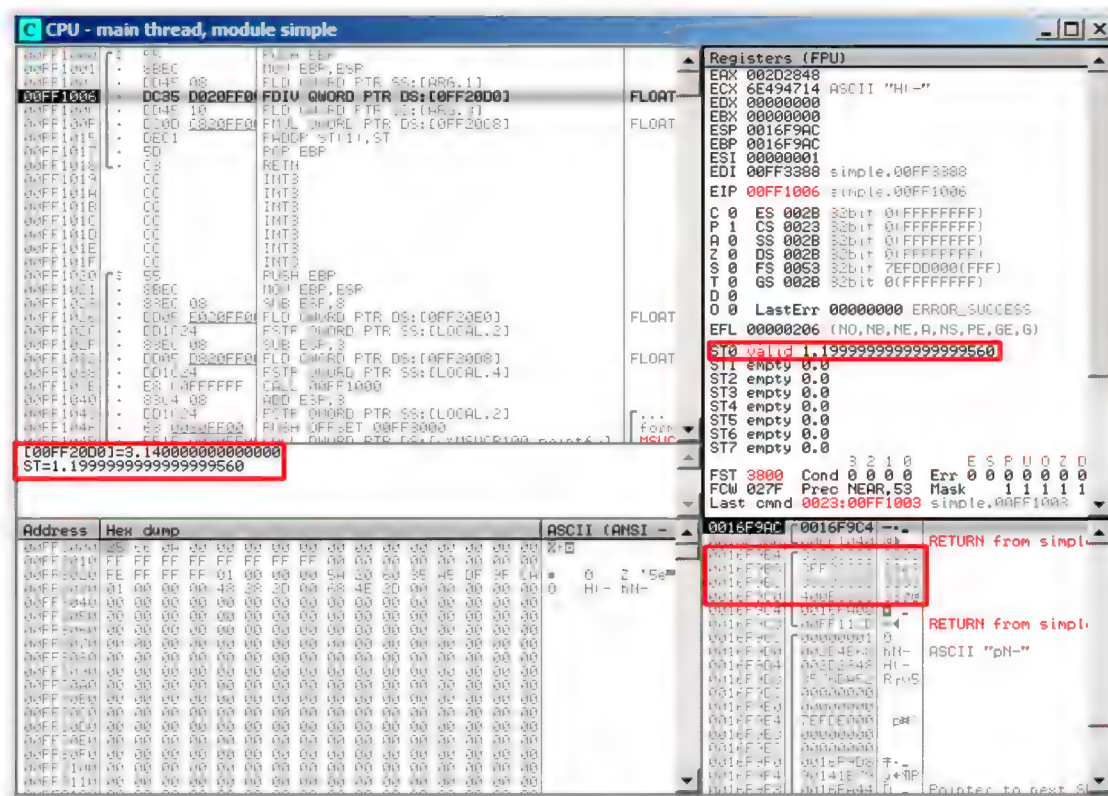


図 1.62: OllyDbg: 最初の FLD が実行された

64ビットIEEE 754浮動小数点から80ビット（FPU内部で使用される）への避けられない変換エラーのため、ここでは1.299に近い1.1999...が見られます。

EIP は次の命令（FDIV）を指すようになり、メモリから倍精度浮動小数点数（定数）がロードされます。便宜上、OllyDbg は3.14を示します。

さらにトレースしましょう。FDIV が実行されましたが、ST(0) に0.382... (商) が含まれています。

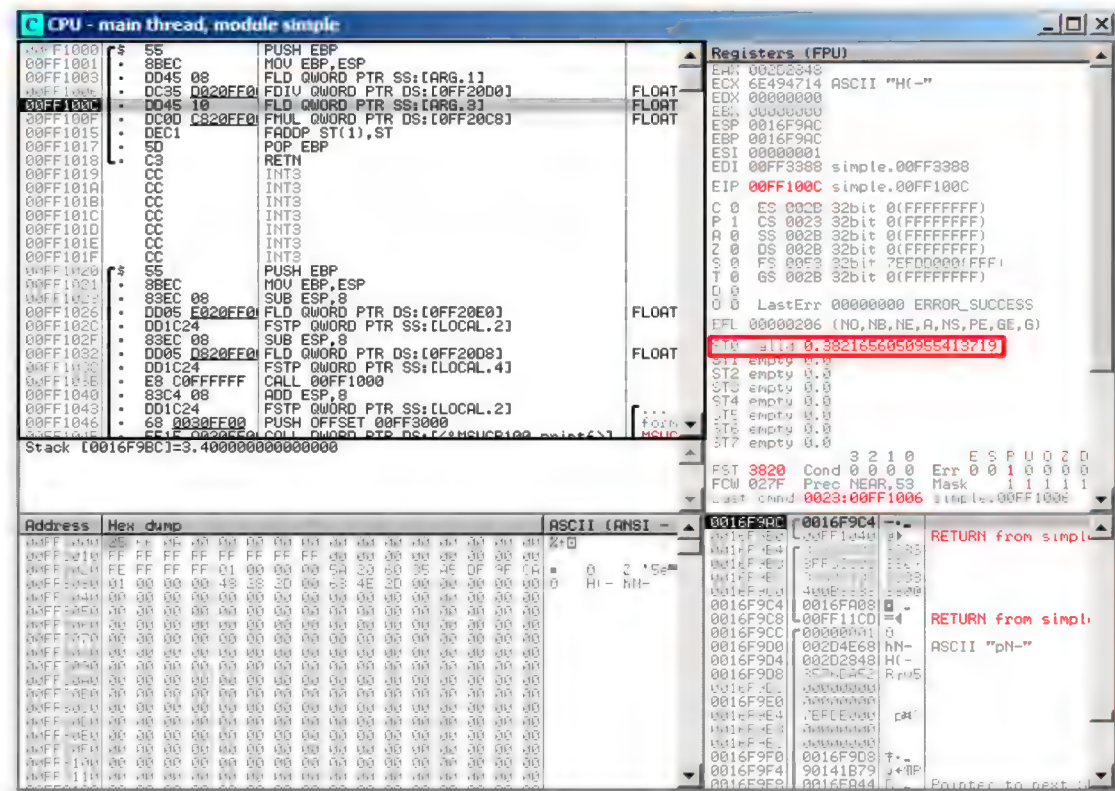


図 1.63: OllDbg: FDIV が実行された

3番目のステップ：次の FLD が実行され、ST(0) に3.4をロードします（ここでは、およそ3.39999...の値が確認できます）。

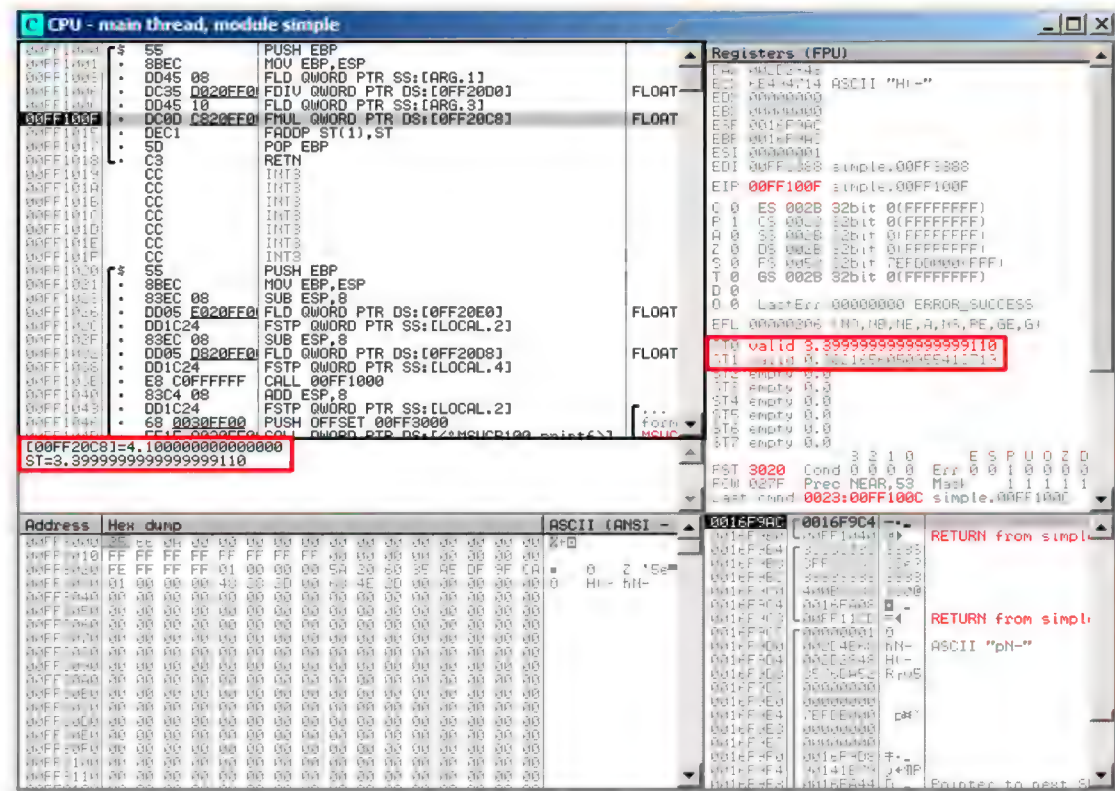


図 1.64: OllyDbg: 2回目の FLD が実行された

同時に、**商**は ST(1) にプッシュされます。今、EIP は次の命令 FMUL を指しています。これは、OllyDbg が示すメモリから定数4.1をロードします。

次：FMUL が実行されたので、積は ST(0) になります。

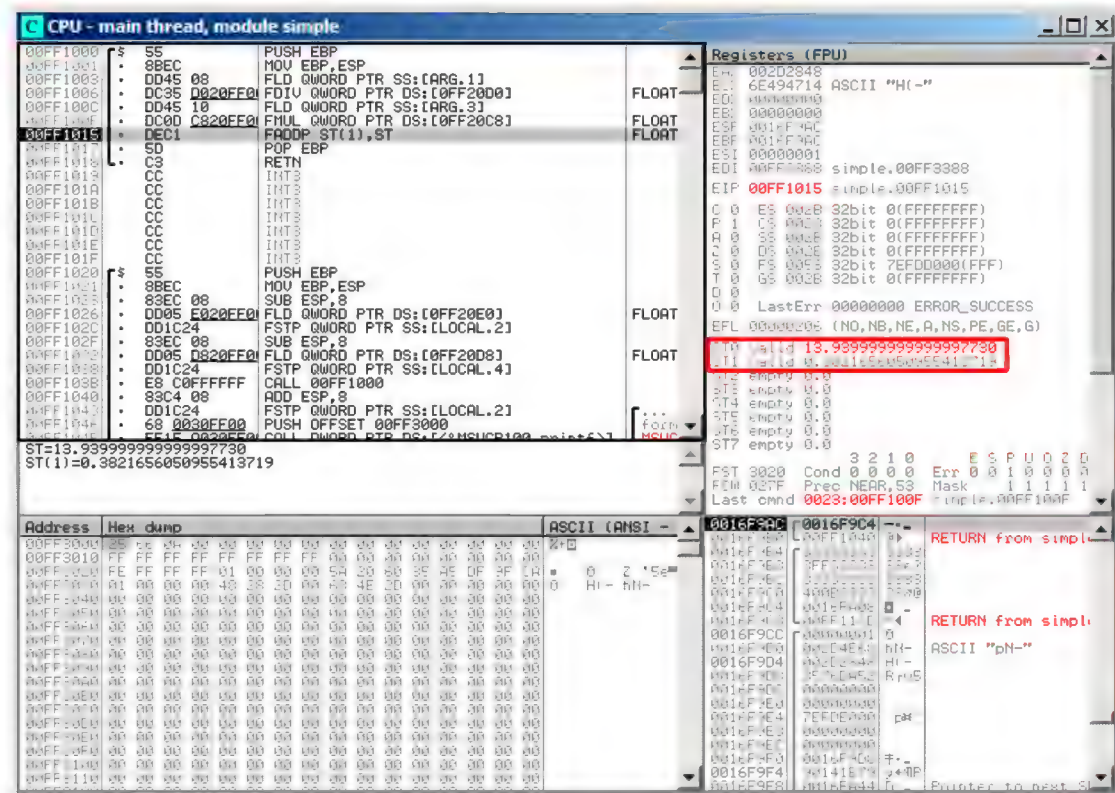


図 1.65: OllyDbg: FMUL が実行された

次に、FADDP が実行され、加算結果が ST(0) になり、ST(1) がクリアされます。

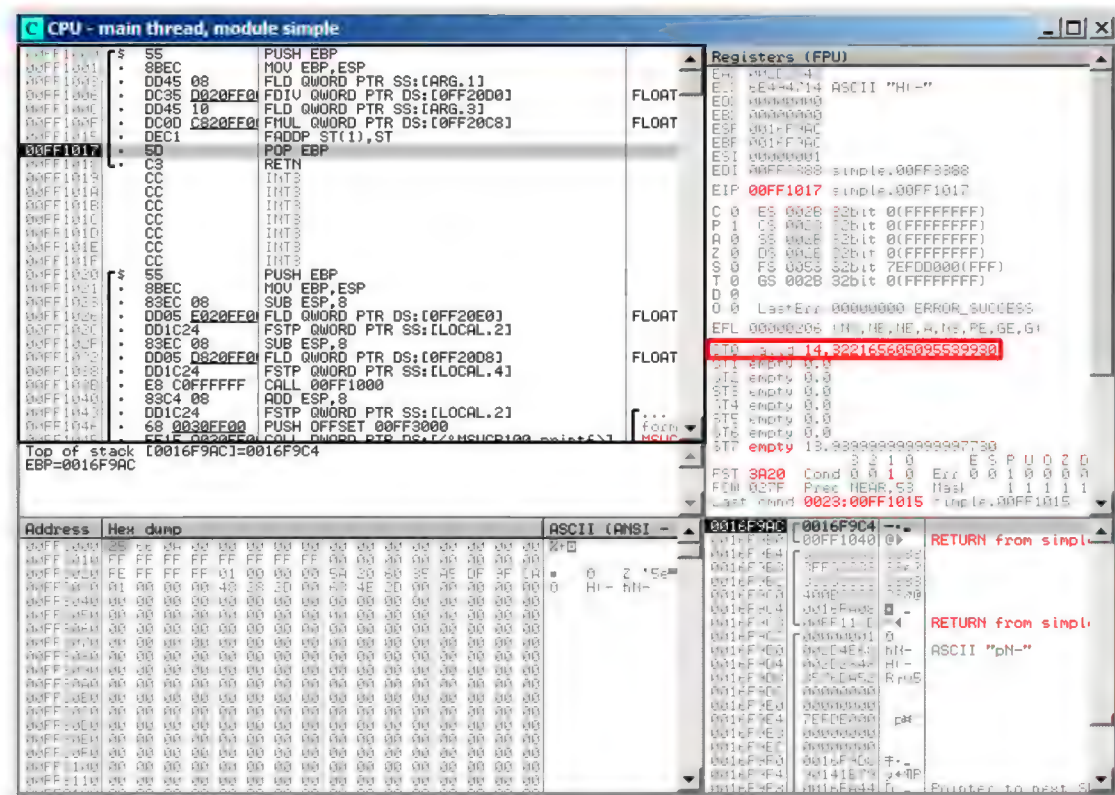


図 1.66: OllyDbg: FADDP が実行された

関数はその値を ST(0) に戻すため、結果は ST(0)に残ります。

main() は後でこの値をレジスタから取得します。

13.93...値は現在 ST(7) に位置しています。どうしてでしょうか？

この本の中で少し前に読んだことがあるように、FPUレジスタはスタックです。1.19.2 on page 268 しかしこれは単純化されています。

説明したようにハードウェアで実装されているとしたら、プッシュとポップ中に7つのレジスタのすべての内容を隣接するレジスタに移動（またはコピー）する必要があります。

現実には、FPUは8つのレジスタと、現在の「トップ・オブ・スタック」であるレジスタ番号を含むポインタ（TOP と呼ばれる）とを持ちます。

値がスタックにプッシュされると、TOP は次に使用可能なレジスタをポイントし、そのレジスタに値が書き込まれます。

値がポップされると、プロシージャは元に戻されますが、解放されたレジスタはクリアされません（クリアされる可能性があります、パフォーマンスが低下する可能性があります）。それがここにある理由です。

FADDP はスタックに合計を保存した後、要素を1つポップしたと言えるでしょう。

しかし、実際には、この命令は合計を保存してから TOP にシフトします。

より正確には、FPUのレジスタは循環バッファです。

GCC

GCC 4.4.1 (-O3 オプション付き) は、わずかに異なる同じコードを出力します：

Listing 1.201: 最適化 GCC 4.4.1

```

f                public f
                proc near
arg_0            = qword ptr 8
arg_8            = qword ptr 10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14
; 現在のスタック状態: ST(0) = 3.14

                mov     ebp, esp
                fdivr   [ebp+arg_0]
; 現在のスタック状態: ST(0) = 除算の結果

                fld     ds:dbl_8048610 ; 4.1
; 現在のスタック状態: ST(0) = 4.1, ST(1) = 除算の結果

                fmul    [ebp+arg_8]
; 現在のスタック状態: ST(0) = 乗算の結果、ST(1) = 除算の結果

                pop     ebp
                faddp   st(1), st
; 現在のスタック状態: ST(0) = 加算の結果

                retn
f                endp

```

違いは、まず3.14がスタック (ST(0)) にプッシュされ、arg_0 の値が ST(0) の値で除算される点です。

FDIVR は、*Reverse Divide* の略で、除数と配当を入れ替えて割ります。同様に乗算命令はありません。これは可換演算であるため、FMUL には-R の部分がなくてもかまいません。

FADDP は2つの値を加算するだけでなく、スタックから値を1つポップします。その操作の後、ST(0) は合計を保持します。

ARM: 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

ARMが標準化された浮動小数点サポートを得るまで、いくつかのプロセッサメーカーは独自の命令拡張を追加しました。次に、VFP (Vector Floating Point) を標準化しました。x86との重要な違いの1つは、ARMではスタックがなく、レジスタだけで動作するという事です。

Listing 1.202: 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

f	VLDR	D16, =3.14	
	VMOV	D17, R0, R1 ; "a" をロード	
	VMOV	D18, R2, R3 ; "b" をロード	
	VDIV.F64	D16, D17, D16 ; a/3.14	
	VLDR	D17, =4.1	
	VMUL.F64	D17, D18, D17 ; b*4.1	
	VADD.F64	D16, D17, D16 ; +	
	VMOV	R0, R1, D16	
	BX	LR	
dbl_2C98	DCFD 3.14		; DATA XREF: f
dbl_2CA0	DCFD 4.1		; DATA XREF: f+10

そこで、ここではDの接頭辞を使用して新しいレジスタをいくつか見ていきます。

これらは64ビットレジスタで、32個あり、浮動小数点数(double)とSIMD(ARMではNEONと呼ばれる)の両方に使用できます。

32ビットの32ビットSレジスタもあり、単精度浮動小数点数(浮動小数点数)として使用されます。

暗記するのは簡単です.Dレジスタは倍精度の数値用であり、Sレジスタは単精度の数値です。詳細は: ?? on page ??

両方の定数(3.14と4.1)はIEEE 754形式でメモリに格納されます。

VLDR と VMOV は、簡単に推測できるように、LDR 命令と MOV 命令に似ていますが、Dレジスタで動作します。

これらの命令は、Dレジスタと同様に、浮動小数点数だけでなく、SIMD (NEON) 演算にも使用でき、これもすぐに表示されることに注意してください。

引数はRレジスタを介して共通の方法で関数に渡されますが、倍精度の各数値のサイズは64ビットなので、各レジスタを渡すには2つのRレジスタが必要です。

VMOV D17, R0, R1 は R0 と R1 から2つの32ビット値を1つの64ビット値に合成し、D17に保存します。

VMOV R0, R1, D16 は逆の演算です。D16にあったものは、R0 と R1 の2つのレジスタに分割されます。これは、格納に64ビット必要な倍精度数が R0 と R1 に返されるためです。

VDIV、VMUL、VADD はそれぞれ商、積、和を計算する浮動小数点数を処理する命令です。

Thumb-2のコードは同じです。

ARM: 最適化 Keil 6/2013 (Thumbモード)

```

f
    PUSH    {R3-R7,LR}
    MOVS    R7, R2
    MOVS    R4, R3
    MOVS    R5, R0
    MOVS    R6, R1
    LDR     R2, =0x66666666 ; 4.1
    LDR     R3, =0x40106666
    MOVS    R0, R7
    MOVS    R1, R4
    BL      __aeabi_dmul
    MOVS    R7, R0
    MOVS    R4, R1
    LDR     R2, =0x51EB851F ; 3.14
    LDR     R3, =0x40091EB8
    MOVS    R0, R5
    MOVS    R1, R6
    BL      __aeabi_ddiv
    MOVS    R2, R7
    MOVS    R3, R4
    BL      __aeabi_dadd
    POP     {R3-R7,PC}

; IEEE 754形式で定数4.1
dword_364    DCD 0x66666666          ; DATA XREF: f+A
dword_368    DCD 0x40106666          ; DATA XREF: f+C
; IEEE 754形式で定数3.14
dword_36C    DCD 0x51EB851F          ; DATA XREF: f+1A
dword_370    DCD 0x40091EB8          ; DATA XREF: f+1C

```

KeilはFPUまたはNEONをサポートしていないプロセッサ用のコードを生成しました。

倍精度浮動小数点数は、汎用Rレジスタを介して渡され、FPU命令の代わりに浮動小数点数の乗算、除算、加算をエミュレートするサービスライブラリ関数（`__aeabi_dmul`、`__aeabi_ddiv`、`__aeabi_dadd` など）が呼び出されます。

もちろん、それはFPUコプロセッサよりも遅いですが、何もないよりはましです。

ところで、同様のFPUエミュレートライブラリは、コプロセッサが貴重で高価で、高価なコンピュータにしかインストールされていなかったx86の世界で非常に人気がありました。

FPUコプロセッサエミュレーションは、ARMワールドではソフトフロートまたは *armel*（エミュレーション）と呼ばれ、コプロセッサのFPU命令はハードフロートまたは *armhf* と呼ばれます。

ARM64: 最適化 GCC (Linaro) 4.9

とってもコンパクトなコードです。

Listing 1.203: 最適化 GCC (Linaro) 4.9

```
f:
```

```

; D0 = a, D1 = b
    ldr    d2, .LC25        ; 3.14
; D2 = 3.14
    fdiv   d0, d0, d2
; D0 = D0/D2 = a/3.14
    ldr    d2, .LC26        ; 4.1
; D2 = 4.1
    fmadd  d0, d1, d2, d0
; D0 = D1*D2+D0 = b*4.1+a/3.14
    ret

; IEEE 754形式の定数
.LC25:
    .word  1374389535        ; 3.14
    .word  1074339512
.LC26:
    .word  1717986918        ; 4.1
    .word  1074816614

```

ARM64: 非最適化 GCC (Linaro) 4.9

Listing 1.204: 非最適化 GCC (Linaro) 4.9

```

f:
    sub    sp, sp, #16
    str    d0, [sp,8]        ; レジスタの保存領域に"a" を保存する
    str    d1, [sp]          ; レジスタの保存領域に"b" を保存する
    ldr    x1, [sp,8]
; X1 = a
    ldr    x0, .LC25
; X0 = 3.14
    fmov   d0, x1
    fmov   d1, x0
; D0 = a, D1 = 3.14
    fdiv   d0, d0, d1
; D0 = D0/D1 = a/3.14

    fmov   x1, d0
; X1 = a/3.14
    ldr    x2, [sp]
; X2 = b
    ldr    x0, .LC26
; X0 = 4.1
    fmov   d0, x2
; D0 = b
    fmov   d1, x0
; D1 = 4.1
    fmul   d0, d0, d1
; D0 = D0*D1 = b*4.1

    fmov   x0, d0
; X0 = D0 = b*4.1
    fmov   d0, x1

```

```

; D0 = a/3.14
    fmov    d1, x0
; D1 = X0 = b*4.1
    fadd    d0, d0, d1
; D0 = D0+D1 = a/3.14 + b*4.1

    fmov    x0, d0 ; \ 冗長符号
    fmov    d0, x0 ; /
    add     sp, sp, 16
    ret
.LC25:
    .word   1374389535      ; 3.14
    .word   1074339512
.LC26:
    .word   1717986918      ; 4.1
    .word   1074816614

```

非最適化 GCCはもっと冗長です。

いくつかの明確に冗長なコード（最後の2つの FM0V 命令）を含む、不要な値のシャッフルが多くあります。おそらく、GCC 4.9はまだARM64コードを生成するのに適していません。

注目すべきことは、ARM64には64ビットのレジスタがあり、Dレジスタには64ビットのレジスタも含まれているということです。

したがって、コンパイラはローカルスタックではなく **GPR** に *double* 型の値を自由に保存できます。これは32ビットCPUでは不可能です。

また、エクササイズとして、FMADD のような新しい命令を導入することなく、この機能を手動で最適化してみることができます。

第1.19.6節

```

#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}

```

x86

(MSVC 2010) で見てみましょう

Listing 1.205: MSVC 2010

```

CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54

```

```

CONST    ENDS

_main    PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8 ; 最初の変数に領域を確保
    fld     QWORD PTR __real@3ff8a3d70a3d70a4
    fstp    QWORD PTR [esp]
    sub     esp, 8 ; 2番目の変数に領域を確保
    fld     QWORD PTR __real@40400147ae147ae1
    fstp    QWORD PTR [esp]
    call    _pow
    add     esp, 8 ; 1つの変数の領域を77戻す77

; ローカルスタックにまだ8バイト空きがある
; 結果がST(0) に

; printf() に渡すために結果をST(0) からローカルスタックに移す:
    fstp    QWORD PTR [esp]
    push    OFFSET $SG2651
    call    _printf
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main    ENDP

```

FLD および FSTP は、データセグメントとFPUスタックとの間の変数を移動します。pow()¹¹⁰はスタックから両方の値をとり、その結果を ST(0) レジスタに返します。printf() はローカルスタックから8バイトを取り出し、double型の変数として解釈します。

ちなみに、メモリ内の値はIEEE 754形式で格納され、pow() もこの形式で格納されているため、値をメモリからスタックに移動するための一対の MOV 命令を使用でき、変換は不要です。これはARMのための次の例で行われます：[1.19.6](#)

ARM + 非最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

```

_main
var_C      = -0xC

        PUSH    {R7,LR}
        MOV     R7, SP
        SUB     SP, SP, #4
        VLDR    D16, =32.01
        VMOV    R0, R1, D16
        VLDR    D16, =1.54
        VMOV    R2, R3, D16
        BLX     _pow
        VMOV    D16, R0, R1
        MOV     R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"

```

¹¹⁰標準的なC関数であり、与えられたべき乗（指数関数）

```

        ADD     R0, PC
        VMOV    R1, R2, D16
        BLX     _printf
        MOVS    R1, 0
        STR     R0, [SP, #0xC+var_C]
        MOV     R0, R1
        ADD     SP, SP, #4
        POP     {R7, PC}

dbl_2F90  DCFD 32.01      ; DATA XREF: _main+6
dbl_2F98  DCFD 1.54      ; DATA XREF: _main+E

```

前に述べたように、64ビットの浮動ポインタ番号はRレジスタのペアで渡されます。

D-レジスタに触れることなく直接Rレジスタに値をロードすることができるので、このコードは少し冗長です（最適化がオフになっているためです）。

したがって、見てきたように、_pow関数は R0 と R1 で最初の引数を受け取り、R2 と R3 で2番目の引数を受け取ります。関数は、その結果を R0 と R1 のままにします。_powの結果は D16 に移動し、次に R1 と R2 のペアで printf() が結果の数値を取得します。

ARM + 非最適化 Keil 6/2013 (ARMモード)

```

_main
    STMFD     SP!, {R4-R6, LR}
    LDR       R2, =0xA3D70A4 ; y
    LDR       R3, =0x3FF8A3D7
    LDR       R0, =0xAE147AE1 ; x
    LDR       R1, =0x40400147
    BL        pow
    MOV       R4, R0
    MOV       R2, R4
    MOV       R3, R1
    ADR       R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
    BL        _2printf
    MOV       R0, #0
    LDMFD     SP!, {R4-R6, PC}

y          DCD 0xA3D70A4      ; DATA XREF: _main+4
dword_520  DCD 0x3FF8A3D7    ; DATA XREF: _main+8
x          DCD 0xAE147AE1    ; DATA XREF: _main+C
dword_528  DCD 0x40400147    ; DATA XREF: _main+10
a32_011_54Lf  DCB "32.01 ^ 1.54 = %lf", 0xA, 0
                                         ; DATA XREF: _main+24

```

Dレジスタはここでは使用されず、Rレジスタのペアのみが使用されます。

ARM64 + 最適化 GCC (Linaro) 4.9

Listing 1.206: 最適化 GCC (Linaro) 4.9

f:

```

        stp    x29, x30, [sp, -16]!
        add    x29, sp, 0
        ldr    d1, .LC1 ; 1.54をD1にロード
        ldr    d0, .LC0 ; 32.01をD0にロード
        bl     pow
; pow() の結果をD0に
        adrp   x0, .LC2
        add    x0, x0, :lo12:.LC2
        bl     printf
        mov    w0, 0
        ldp    x29, x30, [sp], 16
        ret

.LC0:
; IEEE 754形式で32.01
        .word  -1374389535
        .word  1077936455

.LC1:
; IEEE 754形式で1.54
        .word  171798692
        .word  1073259479

.LC2:
        .string "32.01 ^ 1.54 = %lf\n"

```

定数は D0 と D1 にロードされます。pow() は定数をそこから取り出します。結果は、pow() の実行後に D0 に格納されます。変更や移動をせずに printf() に渡す必要があります。これは、printf() は、Xレジスタからの [integral types](#) とポインタとDレジスタからの浮動小数点引数の引数をとります。

第1.19.7節比較の例

これを試してみましょう

```

#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
};

```

機能の単純さにもかかわらず、それがどのように機能するかを理解することは難しいでしょう。

x86**非最適化 MSVC**

MSVC 2010は以下のコードを生成します。

Listing 1.207: 非最適化 MSVC 2010

```

PUBLIC      _d_max
_TEXT      SEGMENT
_a$ = 8                      ; size = 8
_b$ = 16                    ; size = 8
_d_max     PROC
    push    ebp
    mov     ebp, esp
    fld     QWORD PTR _b$[ebp]

; 現在のスタック状態: ST(0) = _b
; _b (ST(0)) と _aを比較し、レジスタをポップ
    fcomp   QWORD PTR _a$[ebp]

; スタックは空

    fnstsw  ax
    test    ah, 5
    jp      SHORT $LN1@d_max

; a>bの場合のみここに来ます

    fld     QWORD PTR _a$[ebp]
    jmp     SHORT $LN2@d_max
$LN1@d_max:
    fld     QWORD PTR _b$[ebp]
$LN2@d_max:
    pop     ebp
    ret     0
_d_max     ENDP

```

FLD は `_b` を `ST(0)` にロードします。

FCOMP は `ST(0)` の値と `_a` の値を比較し、それに応じてFPUステータスワードレジスタの `C3/C2/C0` ビットを設定します。これは、FPUの現在の状態を反映する16ビットのレジスタです。

ビットがセットされると、FCOMP 命令はスタックから1つの変数もポップします。これは、値を比較してスタックを同じ状態にしておく FCOM とは区別されます。

残念ながら、インテルP6 ¹¹¹ より前のCPUには、`C3/C2/C0` ビットをチェックする条件付きジャンプ命令はありません。おそらく、それは歴史の問題です。(思い起こしてみてください: FPUは過去に別のチップでした)

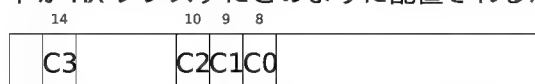
インテルP6で始まる最新のCPUは、FCOMI/FCOMIP/FUCOMI/FUCOMIP 命令を持っていて、同じことをしますが、ZF/PF/CF CPUフラグを変更します。

¹¹¹インテルP6はPentium Pro、Pentium IIなどです。

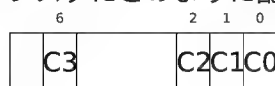
FNSTSW 命令は状態レジスタであるFPUをAXにコピーします。C3/C2/C0 ビットは14/10/8の位置に配置され、AX レジスタの同じ位置にあり、AX —AH の上位部分に配置されます。

- この例では $b > a$ の場合、C3/C2/C0 ビットは0,0,0と設定します。
- $a > b$ の場合、ビットは0,0,1です。
- $a = b$ の場合、ビットは1,0,0です。
- 結果が順序付けられていない場合（エラーの場合）、セットされたビットは1,1,1,1です。

これは、C3/C2/C0 ビットが AX レジスタにどのように配置されるかを示しています。



これは、C3/C2/C0 ビットが AH レジスタにどのように配置されるかを示しています。



test ah, 5¹¹²の実行後、C0 と C2 ビット（0と2の位置）のみが考慮され、他のビットはすべて無視されます。

さて、パリティフラグと注目すべきもう1つの歴史的基礎についてお話ししましょう。

このフラグは、最後の計算結果の1の数が偶数の場合は1に設定され、奇数の場合は0に設定されます。

Wikipedia¹¹³を見てみましょう：

パリティフラグをテストする一般的な理由の1つに、無関係なFPUフラグをチェックすることがあります。FPUには4つの条件フラグ（C0～C3）がありますが、直接テストすることはできず、最初にフラグレジスタにコピーする必要があります。これが起こると、C0はキャリーフラグに、C2はパリティフラグに、C3はゼロフラグに置かれます。C2フラグは、例えば比較できない浮動小数点値（NaNまたはサポートされていない形式）がFUCOM命令と比較されます。

Wikipediaで述べられているように、パリティフラグはFPUコードで使用されることがあります。

C0 と C2 の両方が0に設定されている場合、PF フラグは1に設定されます。その場合、後続の JP (*jump if PF==1*) が実行されます。いろいろな場合の C3/C2/C0 の値を思い出すと、条件ジャンプ JP は、 $b > a$ または $a = b$ の場合に実行されます。（test ah, 5 命令によってクリアされているので、C3 ビットはここでは考慮されていません）

それ以降はすべて簡単です。条件付きジャンプが実行された場合、FLD は ST(0) の `_b` の値をロードし、実行されていないければ `_a` の値をロードします。

¹¹²5=101b

¹¹³https://en.wikipedia.org/wiki/Parity_flag

C2? のチェックは？

C2 フラグはエラー（NaNなど）の場合に設定されますが、コードではチェックされません。プログラマがFPUエラーを気にする場合は、チェックを追加する必要があります。

最初の OllyDbg の例 : $a=1.2$ と $b=3.4$

OllyDbg で例をロードしてみましょう。

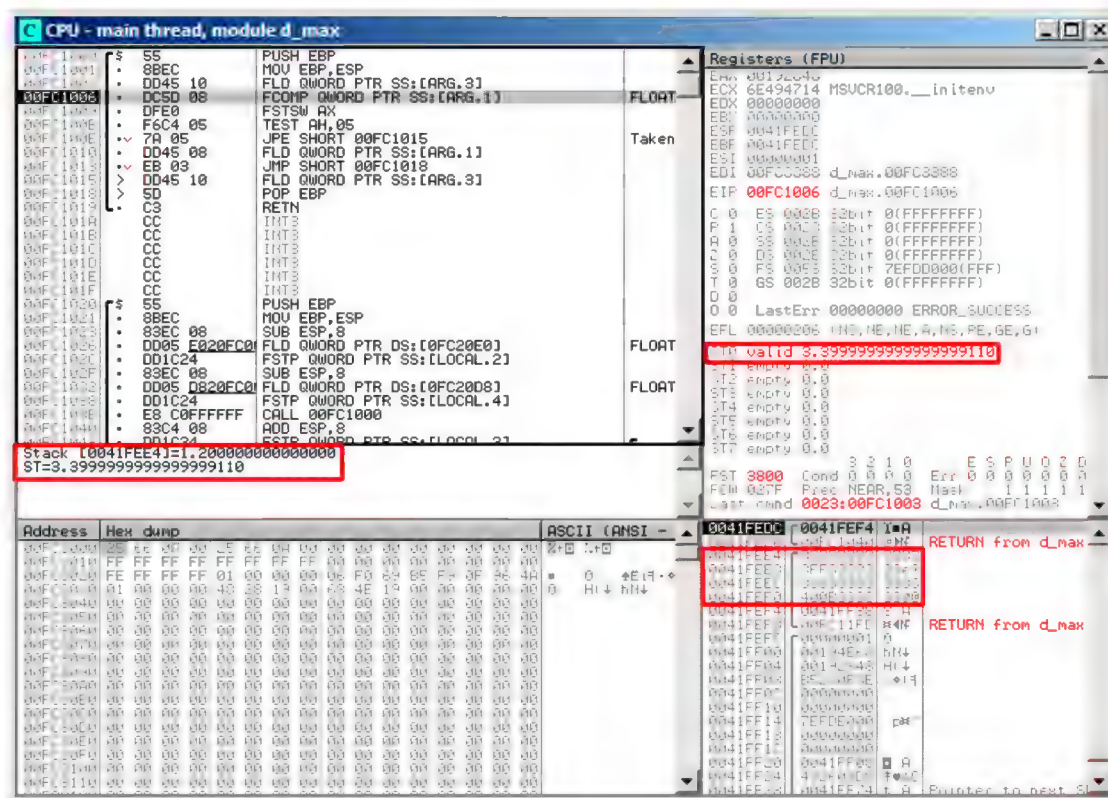


図 1.67: OllyDbg: 最初の FLD が実行される

関数の引数 : $a = 1.2$ と $b = 3.4$ (スタックで見ることができます。32ビット値の2組のペアです) b (3.4) は `ST(0)` にすでにロードされています。そして `FCOMP` が実行されます。OllyDbg は次の `FCOMP` の引数を表示します。スタックにあります。

FCOMP が実行されます。

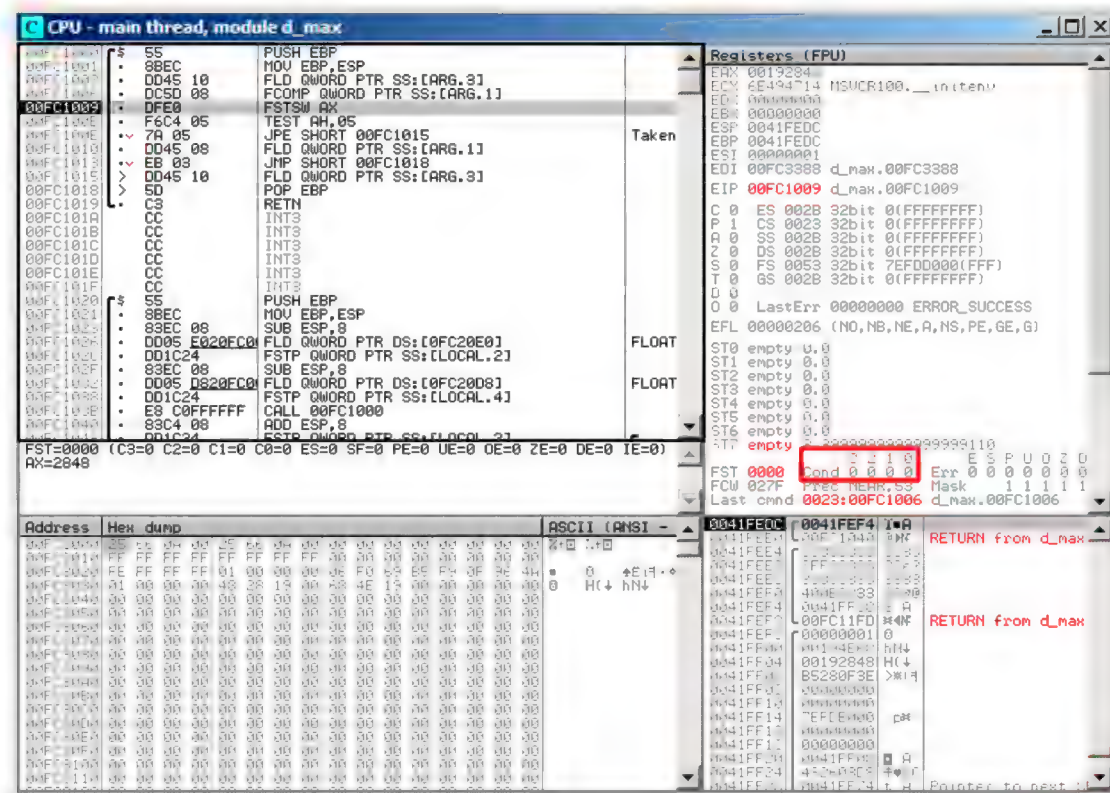


図 1.68: OllyDbg: FCOMP が実行される

FPU条件フラグの状態を見ることができます。すべて0です。ポップされた値は ST(7) に反映されます。この理由については前に書きました：1.19.5 on page 275.

FNSTSW が実行されます。

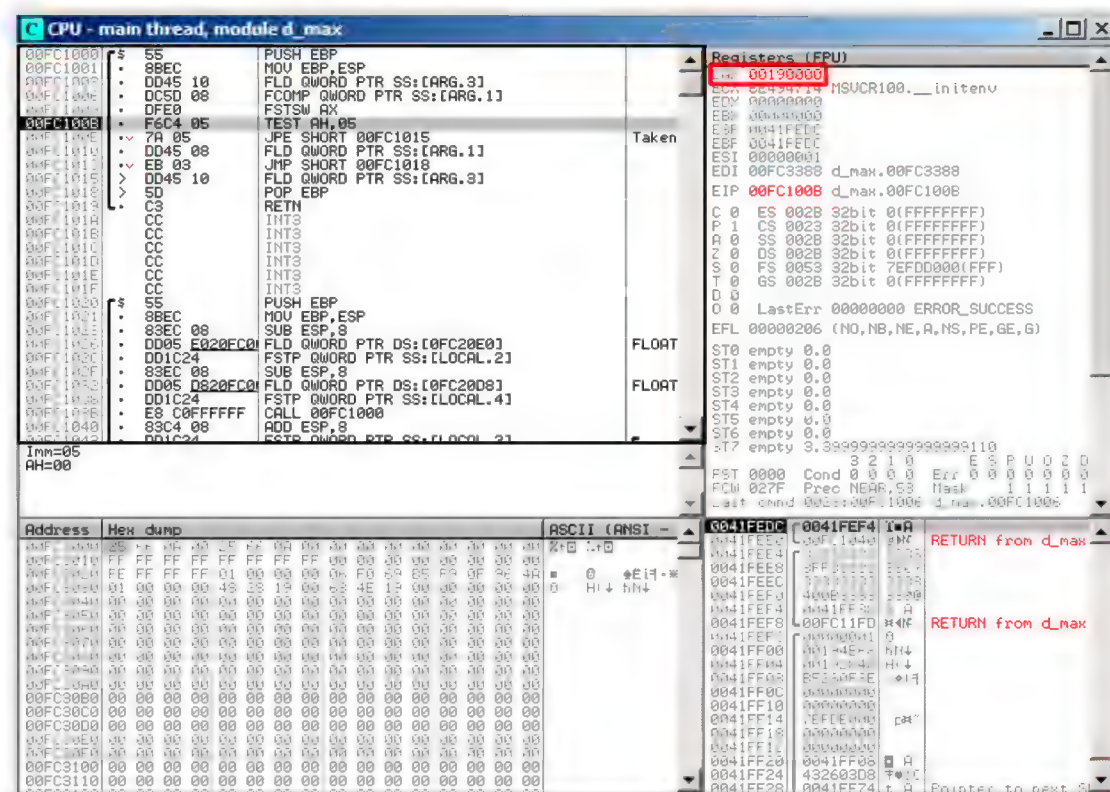


図 1.69: OllyDbg: FNSTSW が実行される

AX レジスタが0であるのが見えます。実際、条件フラグはすべてゼロです。(OllyDbg は FNSTSW 命令を FSTSW としてディスアセンブルします。これは同義語です)

TEST が実行されます。

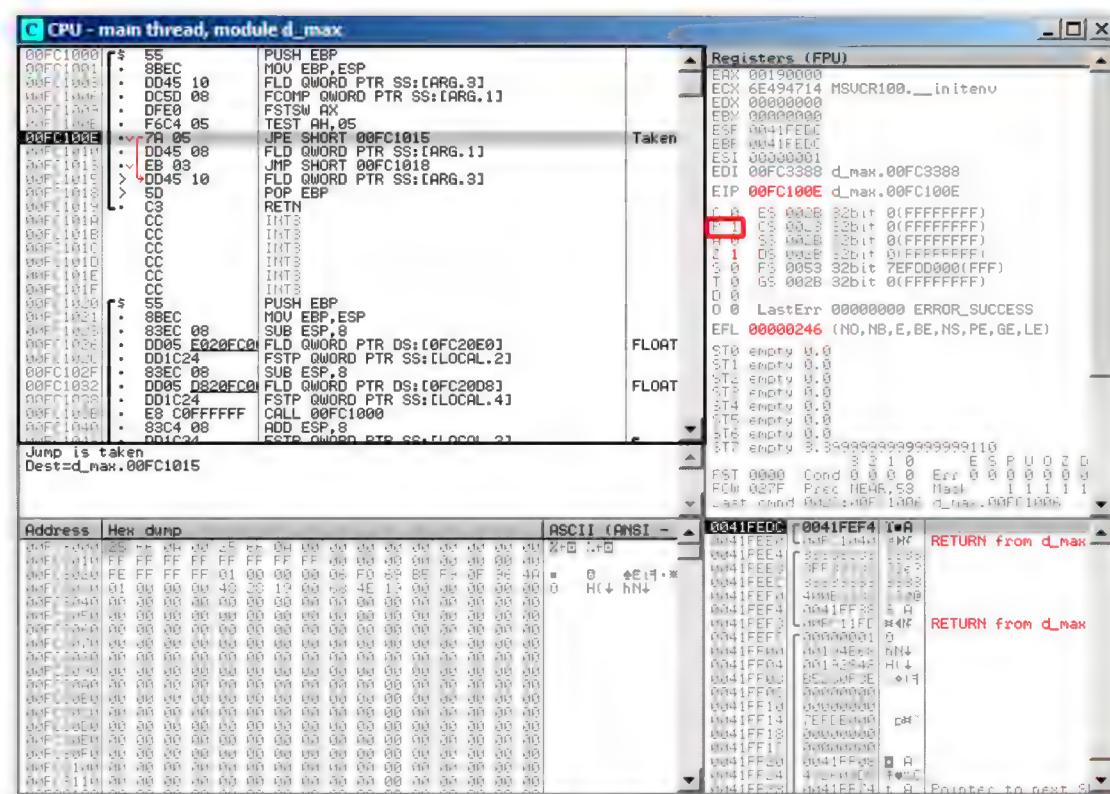


図 1.70: OllyDbg: TEST が実行される

PF フラグが1にセットされます。

実際、0にセットされるビットの数は0で0は偶数です。OllyDbg は JP をJPE¹¹⁴としてデイスアセンブルします。これは同義語です。そして、実行されます。

¹¹⁴Jump Parity Even (x86命令)

JPEが実行され、FLD は *b* (3.4) の値を ST(0) にロードします。

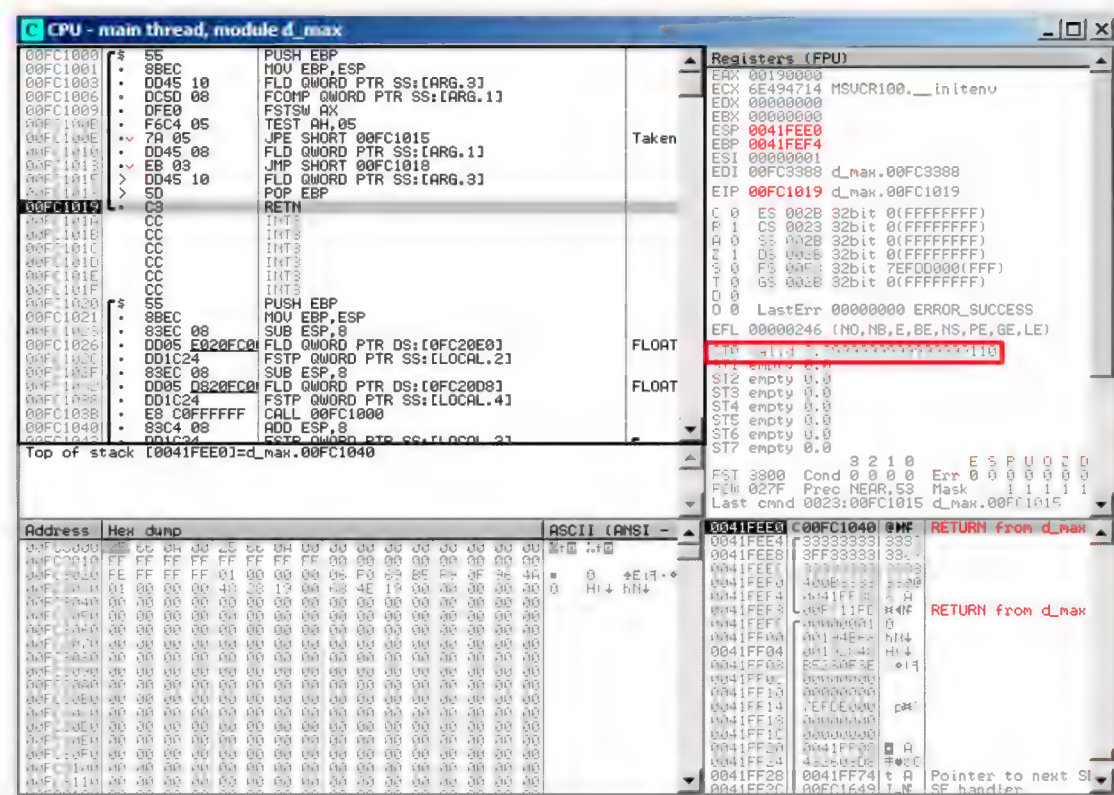


図 1.71: OllyDbg: 次の FLD が実行される

関数が終了します。

The screenshot displays a debugger interface with four main panels:

- Assembly:** Shows assembly instructions for a module named `module_d_max`. Instructions include `PUSH EBP`, `MOV EBP, ESP`, `FLD QWORD PTR SS:[ARG.3]`, `FSTSW AX`, `JPE SHORT 00FC1015`, `FLD QWORD PTR SS:[ARG.1]`, `JMP SHORT 00FC1018`, `FLD QWORD PTR SS:[ARG.3]`, `POP EBP`, `RETN`, `SUB ESP, 8`, `FLD QWORD PTR DS:[0FC20E0]`, `FLD QWORD PTR SS:[LOCAL.2]`, `SUB ESP, 8`, `FLD QWORD PTR DS:[0FC20D8]`, `FSTP QWORD PTR SS:[LOCAL.4]`, `CALL 00FC1000`, `ADD ESP, 8`, and `FSTP QWORD PTR SS:[LOCAL.3]`. Comments like `00FC1000` and `00FC1008` are present.
- Registers (FPU):** Displays the state of CPU registers. `EAX` is `00000009`, `ECX` is `6E445617`, `EDX` is `00220C58`, `EBX` is `00000000`, `ESP` is `0041FEDC`, `EBP` is `0041FEDC`, `ESI` is `00000001`, `EDI` is `00FC3388`, `d_max.00FC3388`, `EIP` is `00FC1006`, `d_max.00FC1006`. FPU registers `C0` through `D0` are shown with their values and precision.
- Stack:** Shows the stack frame. The address `0041FEE4` contains the value `5.6000000000000000`. The address `0041FEE0` contains the value `-4.0000000000000000`. The address `0041FEE8` contains the value `0.0000000000000000`.
- Memory Dump:** Shows a hex dump of memory starting at address `0041FEE0`. The dump shows the values `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00` for the first 16 bytes, followed by `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00` for the next 16 bytes, and so on.

関数の引数: $a = 5.6$ と $b = -4$ $b(-4)$ はすでに ST(0) にロードされています。FCOMP は実行されます。OllYDbg は次の FCOMP の引数を表示します。スタックにあります。

FCOMP が実行されます。

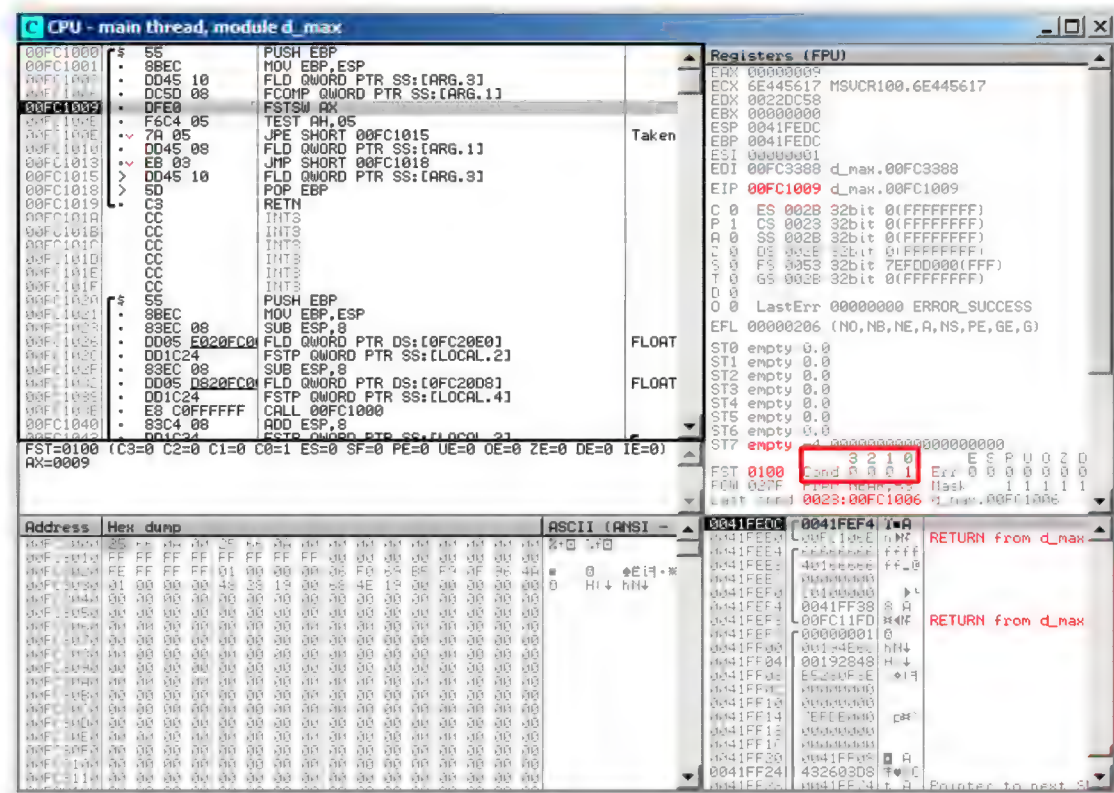


図 1.73: OllyDbg: FCOMP が実行される

FPU条件フラグの状態を見ることができます。C0を除いてすべてゼロです。

FNSTSW が実行されます。

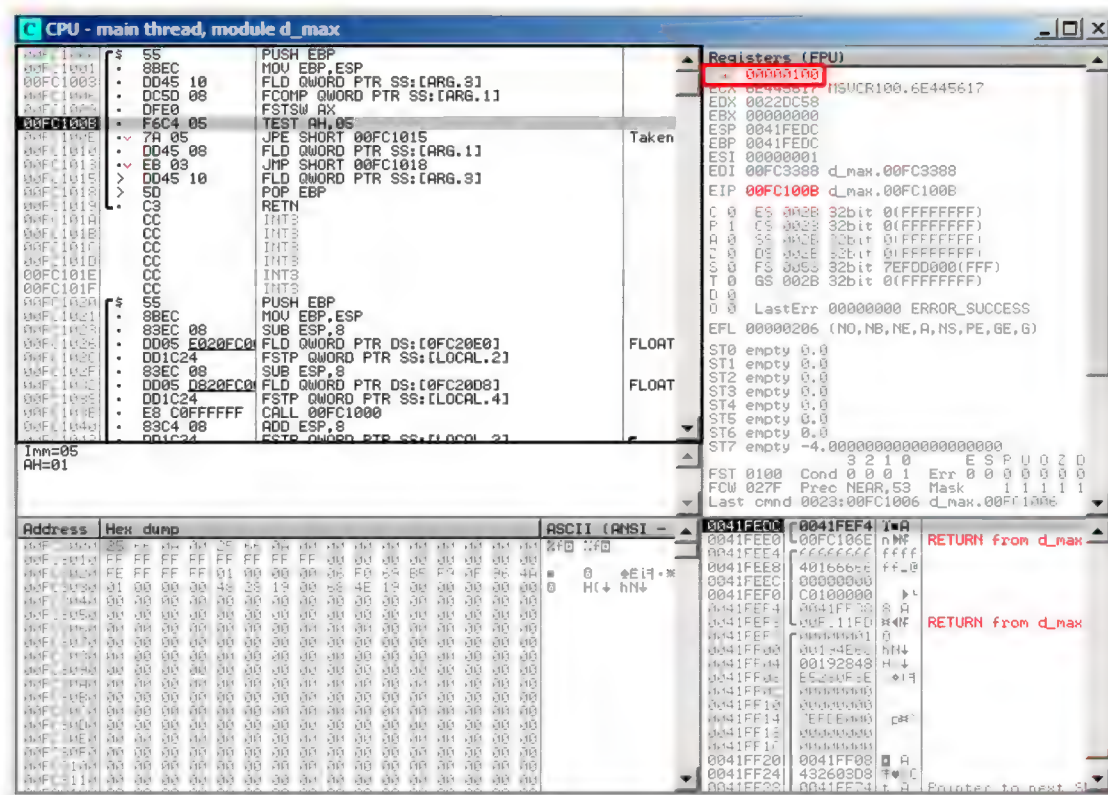


図 1.74: OllyDbg: FNSTSW が実行される

AX レジスタが 0x100 であるのが見えます。C0 フラグは8番目のビットです。

TEST が実行されます。

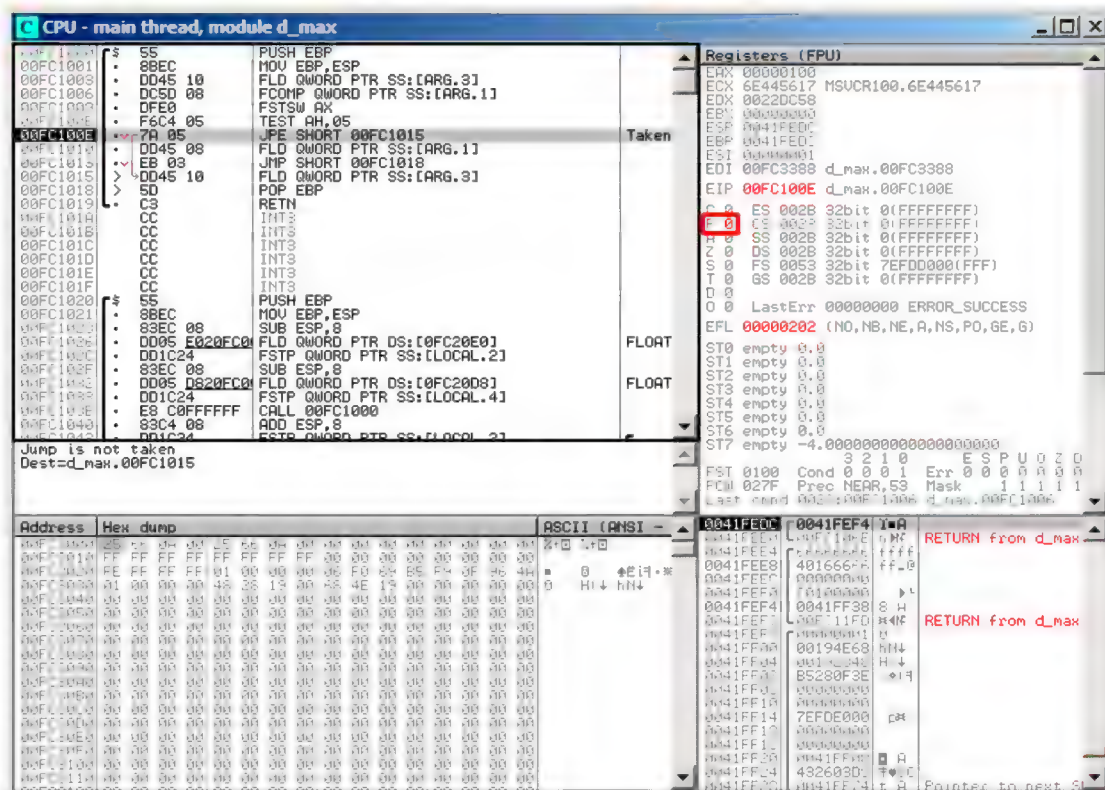


図 1.75: OllyDbg: TEST が実行される

PF フラグがクリアされます。実際、

0x100 にセットされるビットの数は1で、1は奇数です。JPEはスキップされます。

JPEは実行されず、FLD は *a* (5.6) の値を ST(0) にロードします。

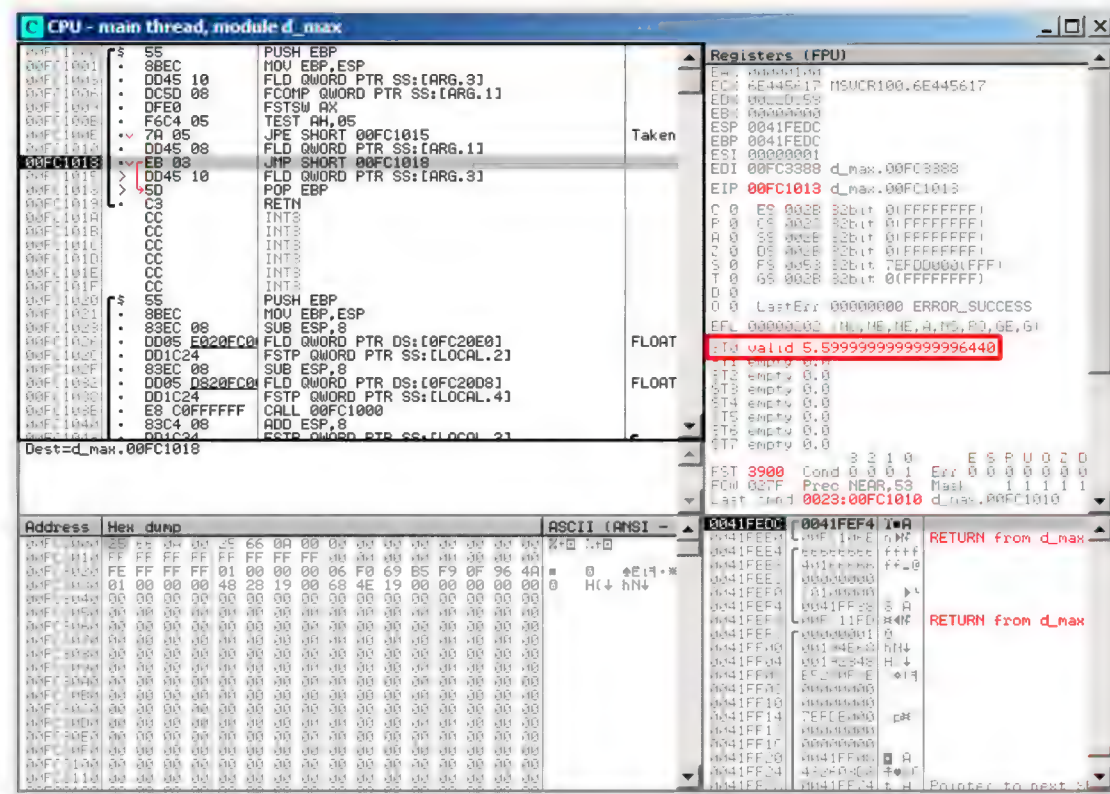


図 1.76: OllyDbg: 次の FLD が実行される

関数が終了します。

最適化 MSVC 2010

Listing 1.208: 最適化 MSVC 2010

```

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max PROC
    fld     QWORD PTR _b$[esp-4]
    fld     QWORD PTR _a$[esp-4]

; 現在のスタック状態: ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; compare _a and ST(1) = (_b)
    fnstsw  ax
    test    ah, 65 ; 00000041H
    jne     SHORT $LN5@_d_max

```

```

; ST(0) を ST(1) にコピーしレジスタをポップ
; (_a) をトップに残す
    fstp    ST(1)

; 現在のスタック状態: ST(0) = _a

    ret     0
$LN5@d_max:
; ST(0) を ST(0) にコピーしレジスタをポップ
; (_b) をトップに残す
    fstp    ST(0)

; 現在のスタック状態: ST(0) = _b

    ret     0
_d_max    ENDP

```

FCOM は、単に値を比較し、FPUスタックを変更しないという点で、FCOMP とは異なります。前の例とは異なり、ここではオペランドは逆順になっています。そのため、C3/C2/C0 の比較結果は異なります。

- この例で $a > b$ の場合、C3/C2/C0 ビットは0,0,0として設定されます。
- $b > a$ の場合、ビットは0,0,1です。
- $a = b$ の場合、ビットは1,0,0です。

test ah, 65 命令は、2ビットの C3 と C0 だけを残します。 $a > b$ の場合は両方ともゼロになります。その場合、JNE ジャンプは実行されません。次に、FSTP ST(1) が続きます。この命令は、ST(0) の値をオペランドにコピーし、FPUスタックから1つの値をポップします。言い換えれば、命令は ST(0) (ここでは _a の値) が ST(1) にコピーされます。その後、_a の2つのコピーがスタックの一番上にあります。次に、1つの値がポップされます。その後、ST(0) には _a が含まれ、機能は終了します。

条件ジャンプ JNE は、 $b > a$ または $a = b$ の2つの場合に実行されます。ST(0) は ST(0) にコピーされ、アイドル (NOP) 操作と同様に、1つの値がスタックからポップされ、スタックの先頭 (ST(0)) には ST(1) 前 (つまり _b) です。その後、関数は終了します。この命令がここで使用される理由は、FPUにスタックから値をポップして破棄するための他の命令がないためです。

最初の OllyDbg の例 : $a=1.2$ と $b=3.4$

FLD が両方とも実行されます。

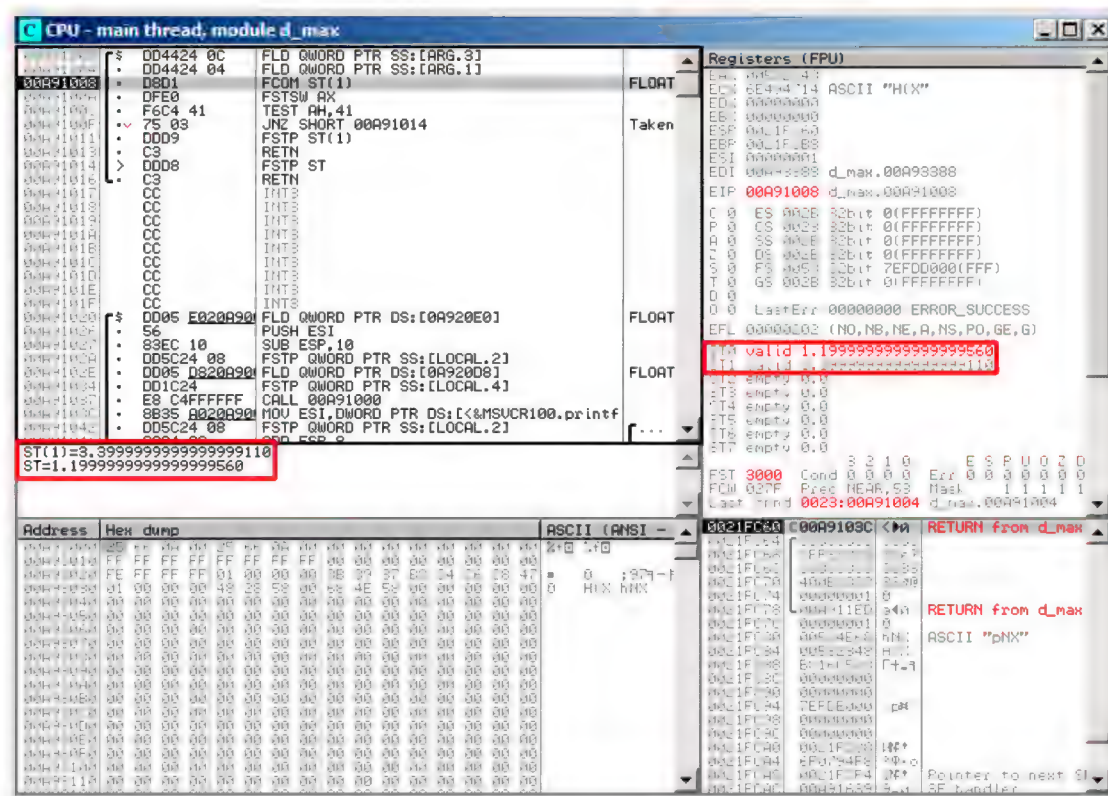


図 1.77: OllyDbg: FLD が両方とも実行される

FCOM が実行されます。OllyDbg は便利なことに、ST(0) と ST(1) の内容を表示します。

FCOM が実行されます。

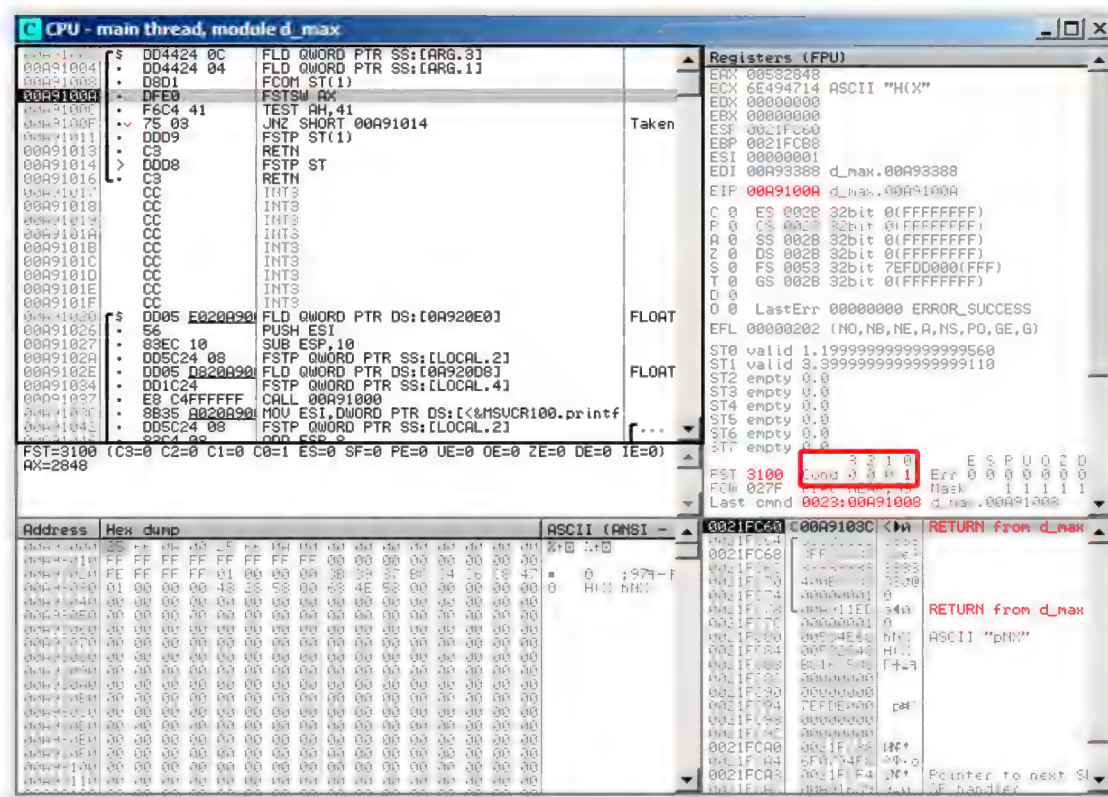


図 1.78: OllyDbg: FCOM が実行される

C0 がセットされ、他の条件フラグはすべてクリアされます。

FNSTSW が実行され、AX=0x3100 になります。

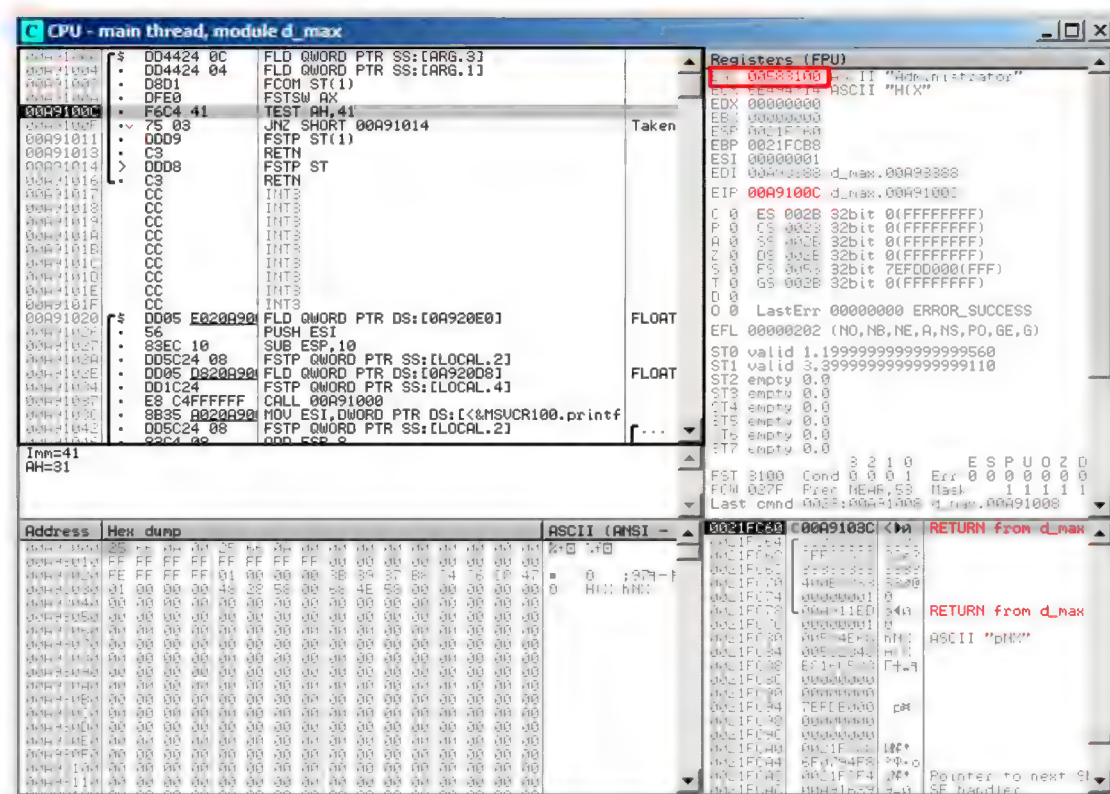


図 1.79: OllyDbg: FNSTSW が実行される

TEST が実行されます。

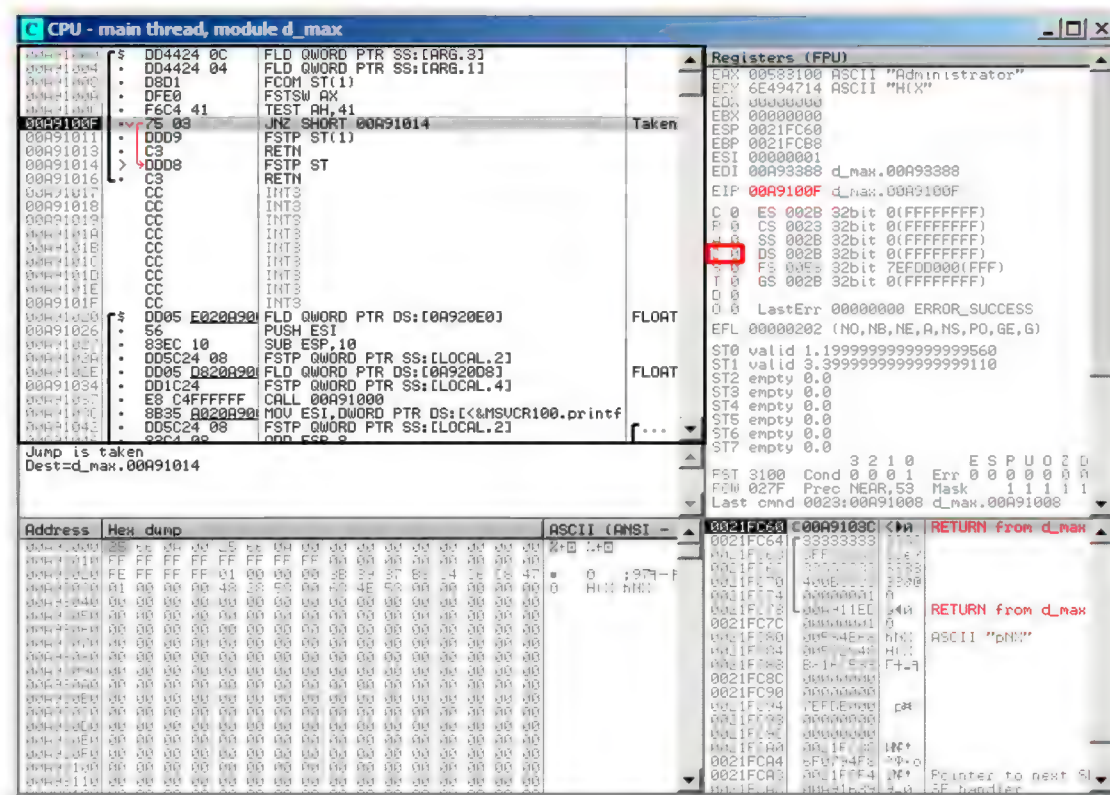


図 1.80: OllyDbg: TEST が実行される

ZF=0の場合、条件ジャンプは実行されます。

FSTP ST (または FSTP ST(0)) が実行されます。1.2がスタックからポップされ、3.4がスタックのトップに残ります。

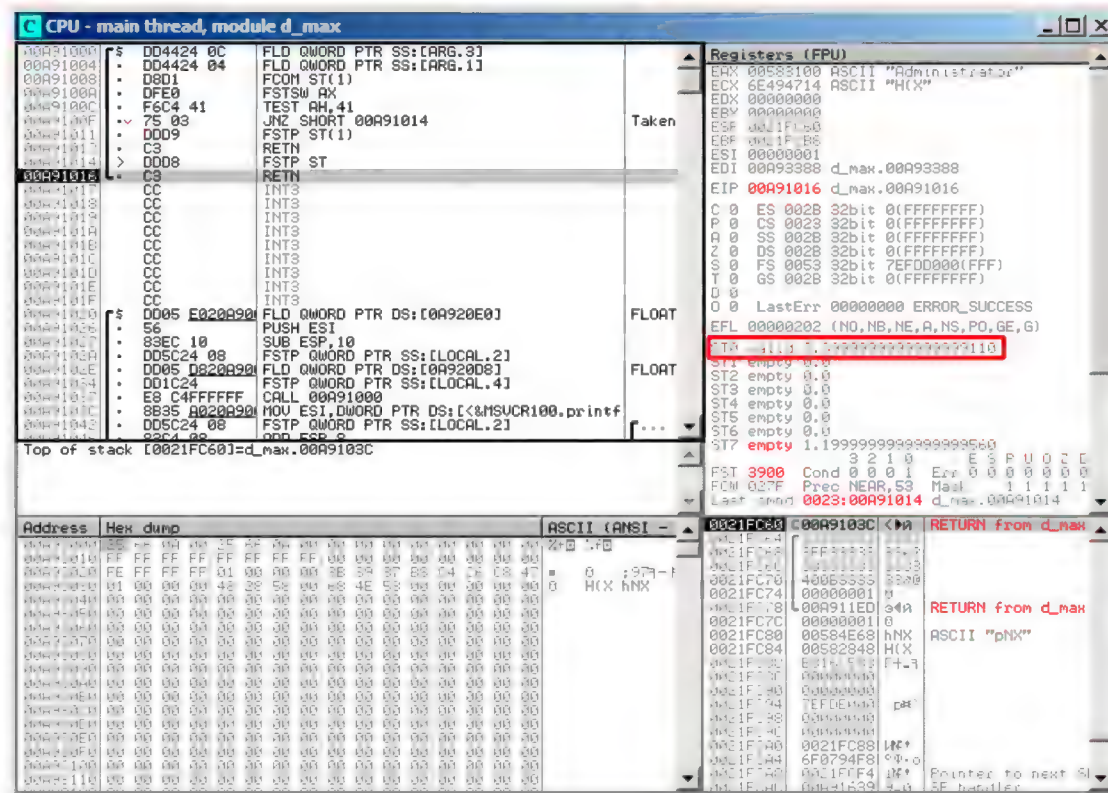


図 1.81: OlllyDbg: FSTP が実行される

FSTP ST を見てみます。

命令は値を1つFPUスタックからポップするだけのようになります。

次の OllyDbg の例 : $a=5.6$ と $b=-4$

FLD が両方とも実行されます。

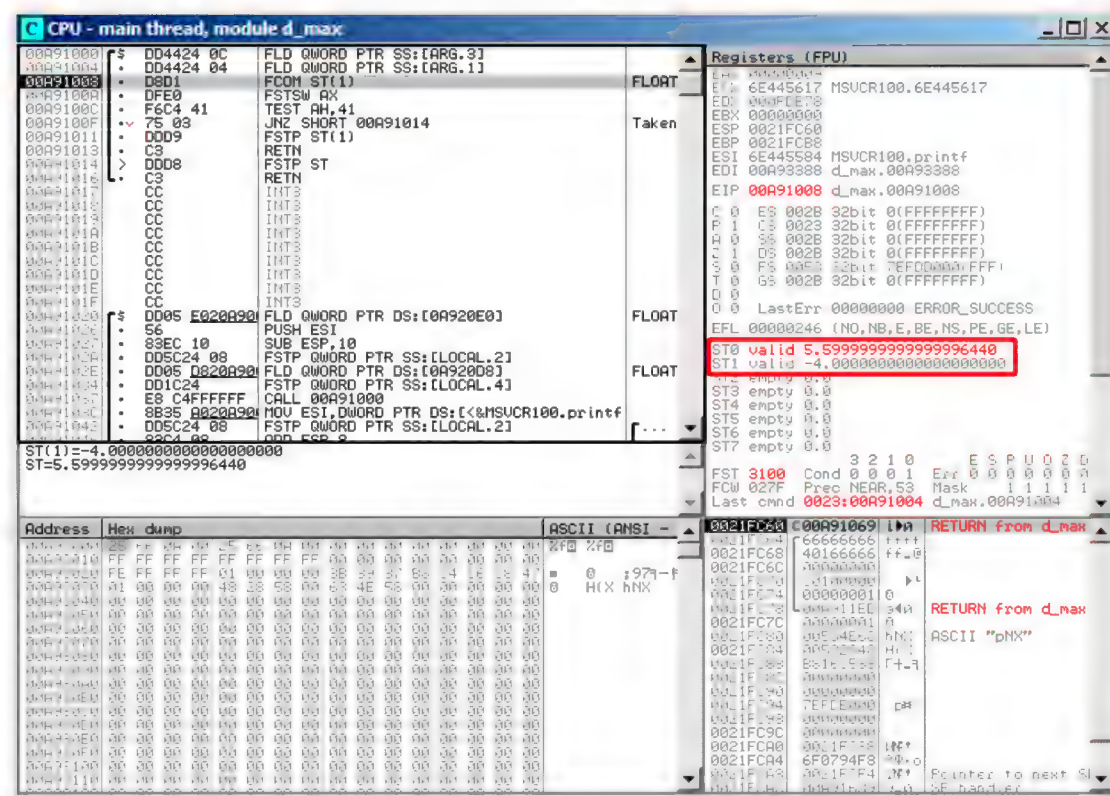


図 1.82: OllyDbg: FLD が両方とも実行される

FLD が実行されます。

FCOM が実行されます。

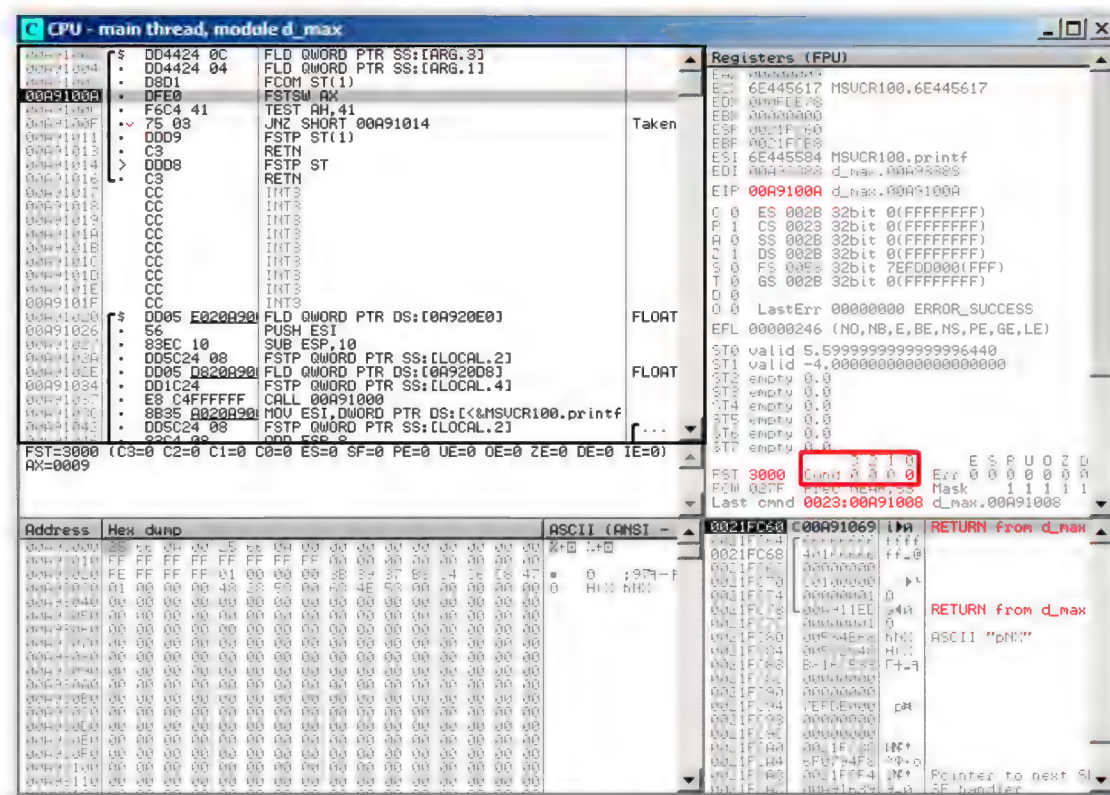


図 1.83: OllyDbg: FCOM が終了する

条件フラグはすべてクリアされます。

FNSTSW が完了し、AX=0x3000 になります。

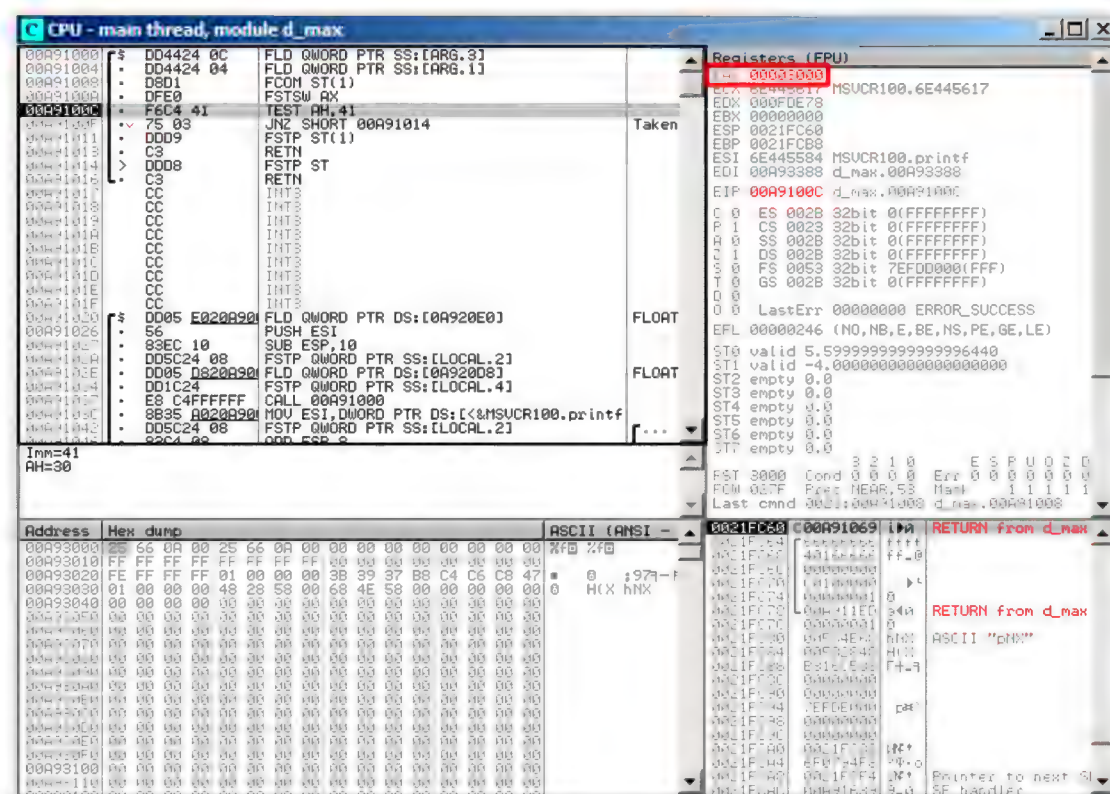


図 1.84: OllyDbg: FNSTSW が実行される

TEST が実行されます。

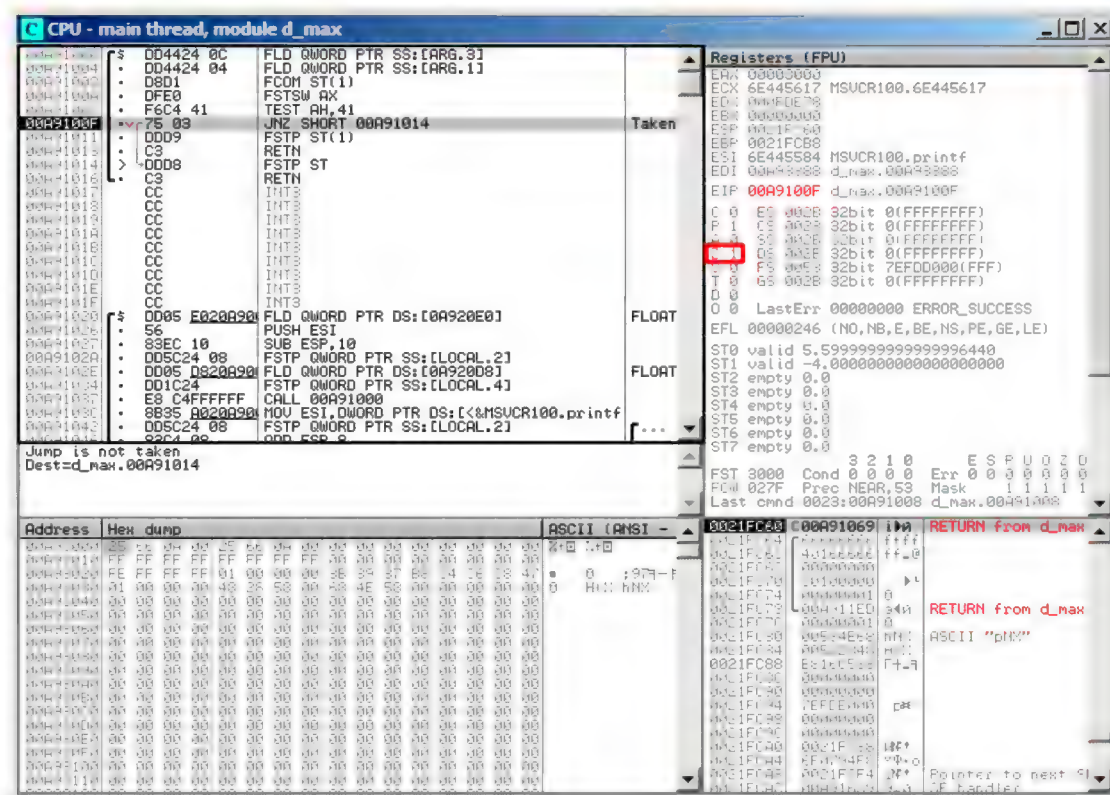


図 1.85: OllDbg: TEST が実行される

ZF=1の場合、ジャンプは発生しません。

FSTP ST(1) が実行されます。5.6はFPUスタックのトップにあります。

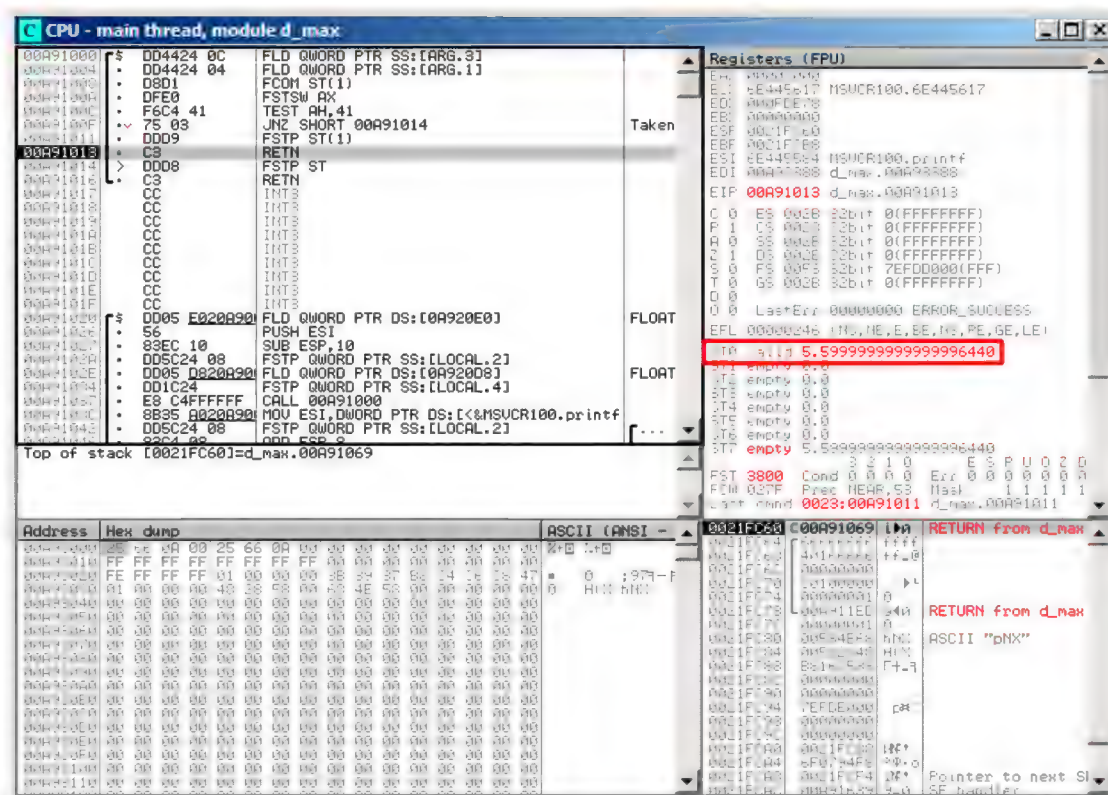


図 1.86: OllyDbg: FSTP が実行される

FSTP ST(1) 命令が以下のように動作することがわかります。値がスタックのトップの残り、ST(1) がクリアされる。

GCC 4.4.1

Listing 1.209: GCC 4.4.1

```
d_max proc near
b          = qword ptr -10h
a          = qword ptr -8
a_first_half = dword ptr 8
a_second_half = dword ptr 0Ch
b_first_half = dword ptr 10h
b_second_half = dword ptr 14h

    push    ebp
    mov     ebp, esp
```

```

    sub     esp, 10h

; aとbをローカルスタックにプッシュ
    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; aとbをFPUスタックにロード

    fld     [ebp+a]
    fld     [ebp+b]

; 現在のスタック状態: ST(0) - b; ST(1) - a

    fxch    st(1) ; this instruction swaps ST(1) and ST(0)

; 現在のスタック状態: ST(0) - a; ST(1) - b

    fucompp ; aとbを比較しスタックから2値をポップ。例: a and b
    fnstsw ax ; FPUステータスをAXに保存
    sahf     ; AHからSF, ZF, AF, PFそしてCFフラグの状態をロード
    setnbe al ; CF=0かつZF=0の場合にALに1を保存
    test     al, al ; AL==0か
    jz       short loc_8048453 ; 真
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]

locret_8048456:
    leave
    retn
d_max endp

```

FUCOMPP は FCOM に似ていますが、スタックから両方の値をポップし、「非数値」を異なる方法で処理します。

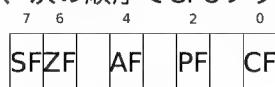
「非数値」について少々。

FPUは、数字でない特別な非数値や **NaN** を扱うことができます。これらは無限大で、0で除算した結果です。非数値は「寡黙」であることも「シグナルを発する」こともできます。「寡黙な」NaNで作業を続行することは可能ですが、「シグナルを発する」NaNで何らかの操作を試みる場合は例外が発生します。

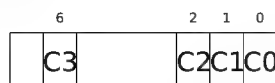
オペランドが**NaN**の場合、FCOM は例外を送出します。FUCOM は、オペランドがシグナルを発する **NaN** (SNaN) である場合にのみ例外を送出します。

次の命令は SAHF (AHをフラグにストア) です。これはFPUに関連しないコードではまれ

な命令です。AHからの8ビットは、次の順序でCPUフラグの下位8ビットに移動します。



FNSTSW が関心のあるビット (C3/C2/C0) を AH に移動し、AH レジスタの位置6,2,0にあることを思い出してください。



言い換えると、fnstsw ax / sahf 命令ペアは、C3/C2/C0 を ZF、PF、および CF に移動します。

異なる条件で C3/C2/C0 の値を思い出してみましょう。

- この例で a が b より大きい場合、C3/C2/C0 は0,0,0に設定されます。
- a が b より小さければ、ビットは0,0,1に設定されます。
- $a = b$ の場合は、1,0,0に設定されます。

言い換えれば、これらのCPUフラグの状態は、3つの FUCOMPP/FNSTSW/SAHF 命令の後に可能になります。

- $a > b$ の場合、CPUフラグは、ZF=0, PF=0, CF=0 として設定されます。
- $a < b$ の場合、フラグは、ZF=0, PF=0, CF=1 として設定されます。
- そして、 $a = b$ ならば、ZF=1, PF=0, CF=0 になります。

CPUのフラグと条件に応じて、SETNBE はALに1または0を格納します。これはほぼ JNBE のものですが、SETcc¹¹⁵ はALに1または0を格納しますが、Jcc は実際にジャンプするかどうかは異なります。SETNBE は、CF=0 および ZF=0 の場合にのみ1を格納します。真でない場合は、0が AL に格納されます。

$a > b$ の場合でのみ、CF と ZF の両方が0になります。

その後、1が AL に格納され、後続の JZ は実行されず、関数は `_a` を返します。それ以外の場合は `_b` が返されます。

最適化 GCC 4.4.1

Listing 1.210: 最適化 GCC 4.4.1

```

d_max      public d_max
            proc near
arg_0      = qword ptr 8
arg_8      = qword ptr 10h

            push    ebp
            mov     ebp, esp
            fld     [ebp+arg_0] ; _a
            fld     [ebp+arg_8] ; _b

```

¹¹⁵cc は 条件コードです

```

; 現在のスタック状態: ST(0) = _b, ST(1) = _a
    fxch    st(1)

; 現在のスタック状態: ST(0) = _a, ST(1) = _b
    fucom   st(1) ; compare _a and _b
    fnstsw  ax
    sahf
    ja      short loc_8048448

; ST(0) にST(0) を保存 (アイドル処理)
; スタックのトップの値をポップ
; _bをトップに残す
    fstp    st
    jmp     short loc_804844A

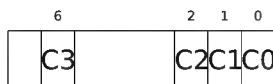
loc_8048448:
; _aをST(1) に保存し、スタックのトップの値をポップし、_aをトップに残す
    fstp    st(1)

loc_804844A:
    pop     ebp
    retn
d_max     endp

```

が SAHF の後に使用されることを除いて、ほとんど同じです。実際には、符号なしの番号比較（これらは JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ）のチェックに「大なり」、「小なり」、「等しい」をチェックする条件ジャンプ命令は CF および ZF フラグが立っているときだけチェックします。

FSTSW/FNSTSW の実行後に、C3/C2/C0 が AH レジスタのどこにあるかを思い出してみましょう。



SAHF の実行後に、AH からのビットが CPU フラグの中にどのようにして保存されるかを思いだしてみましょう。



比較の後、C3 および C0 ビットは ZF および CF に移動するので、条件付きジャンプはその後に働きます。CF と ZF がともにゼロである場合、は実行します。

したがって、ここにリストされている条件付きジャンプ命令は、FNSTSW/SAHF 命令ペアの後で使用できます。

どうやら FPU C3/C2/C0 ステータスビットは、追加の順列を付けずに CPU の基本フラグに簡単にマッピングできるよう、意図的に配置されています。

GCC 4.8.1 with -O3 optimization turned on

いくつかの新しいFPU命令がP6インテルファミリ¹¹⁶に追加されました。これらは FUCOMI (メインCPUのオペランドとフラグの比較) と FCMOVcc (FPOレジスタ上の CMOVcc のように機能します) です。

どうやら、GCCのメンテナは、P6以前のインテルCPU (初期のPentium、80486など) のサポートを中止することに決めました。

また、FPUはP6インテルファミリではもはや別個のユニットではなくなったので、FPUからメインCPUのフラグを変更/チェックすることが可能になりました。

つまり私たちが得るものは次のとおりです。

Listing 1.211: 最適化 GCC 4.8.1

```
fld     QWORD PTR [esp+4]      ; load "a"
fld     QWORD PTR [esp+12]    ; load "b"
; ST0=b, ST1=a
fxch    st(1)
; ST0=a, ST1=b
; "a" と "b" を比較
fucomi  st, st(1)
; a<=bなら、ST1 (ここでは"b") をST0にコピー
; "a" をST0に残す
fcmovbe st, st(1)
; ST1の値を破棄する
fstp    st(1)
ret
```

FXCH (スワップオペランド) がどうしてここにあるのかを推測するのは難しいです。

最初の2つの FLD 命令を交換するか、FCMOVBE (*below or equal*) を FCMOVA (*above*) に置き換えることで、簡単に取り除くことができます。おそらく、それはコンパイラが不正確なためです。

そのため、FUCOMI は ST(0) (*a*) と ST(1) (*b*) を比較し、メインCPUにいくつかのフラグを設定します。FCMOVBE はフラグをチェックし、 $ST0(a) \leq ST1(b)$ なら ST(1) (ここでは *b*) を ST(0) (ここでは *a*) にコピーします。そうでなければ ($a > b$)、ST(0) に *a* を残します。

最後の FSTP は、ST(1) の内容を破棄してスタックの上に ST(0) を残します。

GDBでこの関数をトレースしましょう：

Listing 1.212: 最適化 GCC 4.8.1 and GDB

```
1 dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
2 dennis@ubuntuvm:~/polygon$ gdb d_max
3 GNU gdb (GDB) 7.6.1-ubuntu
4 ...
5 Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols found)...done.
6 (gdb) b d_max
7 Breakpoint 1 at 0x80484a0
8 (gdb) run
```

¹¹⁶Pentium Pro、Pentium-IIなどに始まる

```

9 Starting program: /home/dennis/polygon/d_max
10
11 Breakpoint 1, 0x080484a0 in d_max ()
12 (gdb) ni
13 0x080484a4 in d_max ()
14 (gdb) disas $eip
15 Dump of assembler code for function d_max:
16   0x080484a0 <+0>:      fldl    0x4(%esp)
17 => 0x080484a4 <+4>:      fldl    0xc(%esp)
18   0x080484a8 <+8>:      fxch    %st(1)
19   0x080484aa <+10>:     fucomi %st(1),%st
20   0x080484ac <+12>:     fcmovbe %st(1),%st
21   0x080484ae <+14>:     fstp    %st(1)
22   0x080484b0 <+16>:     ret
23 End of assembler dump.
24 (gdb) ni
25 0x080484a8 in d_max ()
26 (gdb) info float
27   R7: Valid    0x3ffff99999999999800 +1.19999999999999956
28 =>R6: Valid    0x4000d999999999999800 +3.39999999999999911
29   R5: Empty    0x00000000000000000000
30   R4: Empty    0x00000000000000000000
31   R3: Empty    0x00000000000000000000
32   R2: Empty    0x00000000000000000000
33   R1: Empty    0x00000000000000000000
34   R0: Empty    0x00000000000000000000
35
36 Status Word:    0x3000
37                TOP: 6
38 Control Word:   0x037f   IM DM ZM OM UM PM
39                PC: Extended Precision (64-bits)
40                RC: Round to nearest
41 Tag Word:       0x0fff
42 Instruction Pointer: 0x73:0x080484a4
43 Operand Pointer:  0x7b:0xbffff118
44 Opcode:         0x0000
45 (gdb) ni
46 0x080484aa in d_max ()
47 (gdb) info float
48   R7: Valid    0x4000d999999999999800 +3.39999999999999911
49 =>R6: Valid    0x3ffff999999999999800 +1.19999999999999956
50   R5: Empty    0x00000000000000000000
51   R4: Empty    0x00000000000000000000
52   R3: Empty    0x00000000000000000000
53   R2: Empty    0x00000000000000000000
54   R1: Empty    0x00000000000000000000
55   R0: Empty    0x00000000000000000000
56
57 Status Word:    0x3000
58                TOP: 6
59 Control Word:   0x037f   IM DM ZM OM UM PM
60                PC: Extended Precision (64-bits)
61                RC: Round to nearest

```

```

62 Tag Word:          0x0fff
63 Instruction Pointer: 0x73:0x080484a8
64 Operand Pointer:    0x7b:0xbffff118
65 Opcode:            0x0000
66 (gdb) disas $eip
67 Dump of assembler code for function d_max:
68   0x080484a0 <+0>:    fldl    0x4(%esp)
69   0x080484a4 <+4>:    fldl    0xc(%esp)
70   0x080484a8 <+8>:    fxch    %st(1)
71 => 0x080484aa <+10>:   fucomi %st(1),%st
72   0x080484ac <+12>:   fcmovbe %st(1),%st
73   0x080484ae <+14>:   fstp    %st(1)
74   0x080484b0 <+16>:   ret
75 End of assembler dump.
76 (gdb) ni
77 0x080484ac in d_max ()
78 (gdb) info registers
79 eax            0x1      1
80 ecx            0xbffff1c4    -1073745468
81 edx            0x8048340     134513472
82 ebx            0xb7fbf000    -1208225792
83 esp            0xbffff10c    0xbffff10c
84 ebp            0xbffff128    0xbffff128
85 esi            0x0          0
86 edi            0x0          0
87 eip            0x80484ac     0x80484ac <d_max+12>
88 eflags         0x203      [ CF IF ]
89 cs              0x73      115
90 ss              0x7b      123
91 ds              0x7b      123
92 es              0x7b      123
93 fs              0x0        0
94 gs              0x33      51
95 (gdb) ni
96 0x080484ae in d_max ()
97 (gdb) info float
98 R7: Valid      0x4000d99999999999800 +3.39999999999999911
99 =>R6: Valid      0x4000d99999999999800 +3.39999999999999911
100 R5: Empty      0x0000000000000000000
101 R4: Empty      0x0000000000000000000
102 R3: Empty      0x0000000000000000000
103 R2: Empty      0x0000000000000000000
104 R1: Empty      0x0000000000000000000
105 R0: Empty      0x0000000000000000000
106
107 Status Word:    0x3000
108                TOP: 6
109 Control Word:   0x037f    IM DM ZM OM UM PM
110                PC: Extended Precision (64-bits)
111                RC: Round to nearest
112 Tag Word:       0x0fff
113 Instruction Pointer: 0x73:0x080484ac
114 Operand Pointer: 0x7b:0xbffff118

```

```

115 Opcode:                0x0000
116 (gdb) disas $eip
117 Dump of assembler code for function d_max:
118   0x080484a0 <+0>:      fldl   0x4(%esp)
119   0x080484a4 <+4>:      fldl   0xc(%esp)
120   0x080484a8 <+8>:      fxch   %st(1)
121   0x080484aa <+10>:     fucomi %st(1),%st
122   0x080484ac <+12>:     fcmovbe %st(1),%st
123 => 0x080484ae <+14>:     fstp   %st(1)
124   0x080484b0 <+16>:     ret
125 End of assembler dump.
126 (gdb) ni
127 0x080484b0 in d_max ()
128 (gdb) info float
129 =>R7: Valid   0x4000d99999999999800 +3.39999999999999911
130   R6: Empty   0x4000d99999999999800
131   R5: Empty   0x00000000000000000000
132   R4: Empty   0x00000000000000000000
133   R3: Empty   0x00000000000000000000
134   R2: Empty   0x00000000000000000000
135   R1: Empty   0x00000000000000000000
136   R0: Empty   0x00000000000000000000
137
138 Status Word:      0x3800
139                  TOP: 7
140 Control Word:     0x037f   IM DM ZM OM UM PM
141                  PC: Extended Precision (64-bits)
142                  RC: Round to nearest
143 Tag Word:         0x3fff
144 Instruction Pointer: 0x73:0x080484ae
145 Operand Pointer:   0x7b:0xbffff118
146 Opcode:          0x0000
147 (gdb) quit
148 A debugging session is active.
149
150     Inferior 1 [process 30194] will be killed.
151
152 Quit anyway? (y or n) y
153 dennis@ubuntuvms:~/polygon$

```

「ni」を使って、最初の FLD 命令を2つ実行してみましょう。

FPUレジスタを確認してみましょう。(33行目)

以前書いたように、FPUレジスタのセットはスタックではなく循環バッファです。(1.19.5 on page 275) そしてGDBは STx レジスタを表示しませんが、FPUレジスタの内部を表示します。(Rx) (35行目の) 矢印は現在のスタックのトップを示しています。

Status Word(36-37行目) に TOP レジスタの内容を見ることができます。今は6で、スタックのトップは内部レジスタ6を示しています。

a および b の値は FXCH が実行されると交換されます。(54行目)

FUCOMI は実行されます。(83行目) フラグを見てみましょう: CF がセットされます。(95行

目)

FCMOVBE は *b* の値をコピーします。(104行目)

FSTP はスタックのトップの値を1つ残します。(139行目) TOP の値は7で、FPUスタックのトップは内部レジスタ7を示しています。

ARM

最適化 Xcode 4.6.3 (LLVM) (ARMモード)

Listing 1.213: 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

VMOV	D16, R2, R3 ; b
VMOV	D17, R0, R1 ; a
VCMPE.F64	D17, D16
VMRS	APSR_nzcv, FPSCR
VMOVTGT.F64	D16, D17 ; a" をD16にコピー
VMOV	R0, R1, D16
BX	LR

非常に単純なケースです。入力値は D17 および D16 レジスタに格納され、次に VCMPE 命令を使用して比較されます。

コプロセッサ固有のフラグを格納する必要があるため、x86コプロセッサと同様に、ARMコプロセッサには独自のステータスおよびフラグレジスタ (FPSCR¹¹⁷) があります。また、x86と同様に、ARMでは条件付きジャンプ命令がなく、コプロセッサのステータスレジスタ内のビットをチェックできます。したがって、コプロセッサステータスワードからの4ビット (N, Z, C, V) を汎用ステータスレジスタ (APSR¹¹⁸) のビットにコピーするVMRSがあります。

VMOVTGT はDレジスタ用の MOVGT 命令に類似のもので、比較中に一方のオペランドが他方のものより大きい場合に実行されます。(GT—Greater Than)

実行されると、(現在 D17 に格納されている) *a* の値は D16 に書き込まれます。それ以外の場合は、*b* の値は D16 レジスタにとどまります。

最後から2番目の VMOV 命令は、D0レジスタ内の値を R0 および R1 レジスタ対を介して戻すための値を準備します。

最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

Listing 1.214: 最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

VMOV	D16, R2, R3 ; b
VMOV	D17, R0, R1 ; a
VCMPE.F64	D17, D16
VMRS	APSR_nzcv, FPSCR
IT GT	
VMOVTGT.F64	D16, D17

¹¹⁷(ARM) Floating-Point Status and Control Register

¹¹⁸(ARM) Application Program Status Register

VMOV	R0, R1, D16
BX	LR

前の例とほとんど同じですが、少し異なります。すでにわかっているように、ARMモードの多くの命令は条件述語で補うことができます。しかしThumbモードではこのようなことはありません。条件を符号化できる4ビット以上の16ビット命令にはスペースがありません。

ただし、Thumb-2は、古いThumb命令に対する述部を指定できるように拡張されました。ここでは、[IDA](#) で生成されたリストでは、前の例のように VM0VGT 命令が表示されます。

実際、通常の VM0V はそこにエンコードされますが、その直前に IT GT 命令が置かれているため、[IDA](#) は -GT 接尾辞を追加します。

IT 命令は、いわゆる *if-then* ブロックを定義します。

命令の後に最大4つの命令を配置することができ、それぞれに述語接尾辞があります。この例では、GT (*Greater Than*) 条件が真である場合、IT GT は次の命令が実行されることを意味します。

ちなみにAngry Birds (iOS向け) のより複雑なコードの断片は次のとおりです。

Listing 1.215: Angry Birds Classic

```
...
ITE NE
VM0VNE      R2, R3, D16
VM0VEQ      R2, R3, D17
BLX         __objc_msgSend ; not suffixed
...
```

ITE は *if-then-else* を表し、

次の2つの命令の接尾辞をエンコードします。

最初の命令は、ITE でエンコードされた条件 (*NE, not equal*) が真である場合に実行され、2番目の場合は条件が真でない場合に実行されます。(NE の逆条件は EQ (等しい) です。

2番目の VM0V (または VM0VEQ) が通常のもので、接尾辞 (BLX) ではない命令の後に続きます。

もう1つはやや難しく、これもAngry Birdsからです。

Listing 1.216: Angry Birds Classic

```
...
ITTTT EQ
MOVEQ       R0, R4
ADDEQ       SP, SP, #0x20
POPEQ.W     {R8,R10}
POPEQ       {R4-R7,PC}
BLX         __stack_chk_fail ; not suffixed
...
```

命令ニーモニック内の4つの「T」記号は、条件が真である場合に4つの後続の命令が実行されることを意味します。

これが IDA がそれぞれに -EQ サフィックスを追加する理由です。

たとえば、ITEEE EQ (*if-then-else-else-else*) があった場合、接尾辞は次のように設定されます。

```
-EQ
-NE
-NE
-NE
```

Angry Birdsから別の断片です。

Listing 1.217: Angry Birds Classic

```
...
CMP.W          R0, #0xFFFFFFFF
ITTE LE
SUBLE.W        R10, R0, #1
NEGLE         R0, R0
MOVGT         R10, R0
MOVS          R6, #0          ; not suffixed
CBZ           R0, loc_1E7E32 ; not suffixed
...
```

ITTE (*if-then-then-else*) は、LE (*Less or Equal*) 条件が真であれば第1および第2の命令が実行されることを意味し、逆条件 (GT—*Greater Than*) が真であれば第3の命令を実行することを意味します。

コンパイラは通常、可能な組み合わせのすべてを生成しません。

たとえば、前述のAngry Birdsゲーム (iOSのクラシックバージョン) では、IT 命令の変種である、ITE、ITT、ITTE、ITTT、ITTTT のみが使用されます。これらを知るのはどうしたらいいのでしょうか？IDA ではリストファイルを作成することができるため、各オペコードに4バイトを表示するオプションで作成されました。次に、16ビットのオペコードの大部分 (IT は 0xBF) を知っているの、grep を使って次のことを行います。

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

ところで、ARMアセンブリ言語でThumb-2モードを手動でプログラムし、条件付きサフィックスを追加すると、アセンブラは必要な場所に必要なフラグを自動的に IT 命令に追加します。

非最適化 Xcode 4.6.3 (LLVM) (ARMモード)

Listing 1.218: 非最適化 Xcode 4.6.3 (LLVM) (ARMモード)

```
b          = -0x20
a          = -0x18
val_to_return = -0x10
saved_R7   = -4

          STR          R7, [SP,#saved_R7]!
          MOV          R7, SP
```

	SUB	SP, SP, #0x1C
	BIC	SP, SP, #7
	VMOV	D16, R2, R3
	VMOV	D17, R0, R1
	VSTR	D17, [SP,#0x20+a]
	VSTR	D16, [SP,#0x20+b]
	VLDR	D16, [SP,#0x20+a]
	VLDR	D17, [SP,#0x20+b]
	VCMP.E.F64	D16, D17
	VMRS	APSR_nzcv, FPSCR
	BLE	loc_2E08
	VLDR	D16, [SP,#0x20+a]
	VSTR	D16, [SP,#0x20+val_to_return]
	B	loc_2E10
loc_2E08	VLDR	D16, [SP,#0x20+b]
	VSTR	D16, [SP,#0x20+val_to_return]
loc_2E10	VLDR	D16, [SP,#0x20+val_to_return]
	VMOV	R0, R1, D16
	MOV	SP, R7
	LDR	R7, [SP+0x20+b], #4
	BX	LR

既に見たのとほぼ同じですが、 a と b の変数がローカルスタックに格納され、戻り値も格納されるため、冗長コードがとて多くなっています。

最適化 Keil 6/2013 (Thumbモード)

Listing 1.219: 最適化 Keil 6/2013 (Thumbモード)

	PUSH	{R3-R7, LR}
	MOVS	R4, R2
	MOVS	R5, R3
	MOVS	R6, R0
	MOVS	R7, R1
	BL	__aeabi_cdrcmple
	BCS	loc_1C0
	MOVS	R0, R6
	MOVS	R1, R7
	POP	{R3-R7, PC}
loc_1C0	MOVS	R0, R4
	MOVS	R1, R5
	POP	{R3-R7, PC}

ターゲットCPUでサポートされるかに依存できず、また、簡単なビット単位の比較では実行できないため、KeilはFPU命令を生成しません。したがって、外部ライブラリ関数を呼び出して比較を行います：__aeabi_cdrcmple

注意：比較の結果はこの関数によってフラグに残されます。したがって、次の BCS (Carry set—Greater than or equal) 命令は追加コードなしで動作します。

ARM64

最適化 GCC (Linaro) 4.9

```
d_max:
; D0 - a, D1 - b
    fcmpe    d0, d1
    fcsel    d0, d0, d1, gt
; 結果がD0にある
    ret
```

ARM64 ISAには、便宜上、FPSCRの代わりにCPUフラグをAPSRに設定するFPU命令があります。FPUはもはや別個のデバイスではありません。(少なくとも論理的には) ここではFCMPEを参照してください。D0 と D1 (関数の第1引数と第2引数) で渡された2つの値を比較し、APSRフラグ (N, Z, C, V) を設定します。

FCSEL (Floating Conditional Select) は、条件 (GT—Greater Than) に応じて D0 または D1 の値を D0 にコピーし、再びFPSCRの代わりにAPSRレジスタのフラグを使用します。

これは、古いCPUの命令セットに比べてはるかに便利です。

条件が真 (GT) の場合、D0 の値が D0 にコピーされます。(つまり何も起こりません) 条件が真でない場合、D1 の値が D0 にコピーされます。

非最適化 GCC (Linaro) 4.9

```
d_max:
; "Register Save Area" に入力引数を保存
    sub     sp, sp, #16
    str     d0, [sp,8]
    str     d1, [sp]
; 値をリロード
    ldr     x1, [sp,8]
    ldr     x0, [sp]
    fmov    d0, x1
    fmov    d1, x0
; D0 - a, D1 - b
    fcmpe    d0, d1
    ble     .L76
; a>bなら、D0 (a) をX0にロード
    ldr     x0, [sp,8]
    b       .L74
.L76:
; a<=bならD1 (b) をX0にロード
    ldr     x0, [sp]
.L74:
; 結果がX0に
```

```

    fmov    d0, x0
; 結果がD0に
    add     sp, sp, 16
    ret

```

非最適化GCCはより冗長です。

まず、関数は入力引数の値をローカルスタック (*Register Save Area*) に保存します。次に、これらの値をレジスタ D0/D1 にリロードし、最終的に X0/X1 にコピーして FCMPE を使用して比較します。冗長なコードがたくさんありますが、最適化されていないコンパイラの仕組みです。FCMPE は値を比較し、[APSR](#) フラグを設定します。現時点では、コンパイラは、より便利な FCSEL 命令についてはまだ考えていないため、古いメソッドを使用して処理を進めます。つまり、BLE 命令を使用します (*Branch if Less than or Equal*)。最初のケース ($a > b$) では、 a の値が X0 にロードされます。それ以外の場合 ($a \leq b$)、 b の値は X0 にロードされます。最後に、戻り値がこのレジスタにある必要があるため、X0 からの値が D0 にコピーされます。

練習問題

練習として、冗長な命令を削除し、新しい命令 (FCSEL を含む) を導入しないで手動でこのコードを最適化することができます。

最適化 GCC (Linaro) 4.9—float

double の代わりに *float* を使うようにこの例を書き直しましょう。

```

float f_max (float a, float b)
{
    if (a>b)
        return a;

    return b;
};

```

```

f_max:
; S0 - a, S1 - b
    fcmpe   s0, s1
    fcsl    s0, s0, s1, gt
; 結果がS0にある
    ret

```

これは同じコードですが、D-レジスタの代わりにS-レジスタが使用されています。これは、浮動小数点数が32ビットSレジスタ（実際には64ビットDレジスタの下位部分）に渡されるためです。

MIPS

MIPSプロセッサのコプロセッサには条件ビットがあり、これをFPUにセットしてCPUでチェックすることができます。

以前のMIPSには1つの条件ビット（FCC0と呼びます）があり、後のモデルには8つのビット（FCC7-FCC0と呼びます）があります。

このビット（または複数のビット）はFCCRと呼ばれるレジスタに配置されています。

Listing 1.220: 最適化 GCC 4.4.5 (IDA)

```
d_max:
; $f14<$f12 (b<a) なら、FPU条件ビットを設定
    c.lt.d  $f14, $f12
    or      $at, $zero ; NOP
; 条件ビットがセットされていたら、locret_14にジャンプ
    bc1t    locret_14
; この命令は常に実行されます（戻り値に"a" を設定）
    mov.d   $f0, $f12 ; branch delay slot
; この命令は分岐が実行されなかったときのみ実行されます（例えばb>=aの場合）
; 戻り値に"b" を設定
    mov.d   $f0, $f14

locret_14:
    jr      $ra
    or      $at, $zero ; 分岐遅延スロット, NOP
```

C.LT.D は2つの値を比較します。LT は「Less Than」の条件です。D は *double* 型の値を意味します。比較の結果に応じて、FCC0条件ビットはセットまたはクリアされます。

BC1T checks the FCC0 bit and jumps if the bit is set. T means that the jump is to be taken if the bit is set (「True」). There is also the instruction BC1F which jumps if the bit is cleared (「False」).

BC1T はFCC0ビットをチェックし、ビットがセットされていればジャンプします。T は、ビットがセット (「True」) されている場合にジャンプが行われることを意味します。ビットがクリアされるとジャンプする BC1F 命令もあります。(「False」)

ジャンプに応じて、関数引数の1つが \$F0 に配置されます。

第1.19.8節いくつかの定数

IEEE 754でエンコードされた数のWikipediaでいくつかの定数の表現を見つけるのは簡単です。IEEE 754の0.0は、32ビットのゼロビット（単精度の場合）または64ビットのゼロビット（倍精度の場合）として表されることは興味深いことです。したがって、浮動小数点変数をレジスタまたはメモリで0.0に設定するには、MOV または XOR reg, reg 命令を使用します。これは、さまざまなデータ型の多くの変数が存在する構造に適しています。通常のmemset() 関数では、すべての整数変数を0に、すべてのブール変数を *false* に、すべてのポインタをNULLに、すべての浮動小数点変数（任意の精度）を0.0に設定できます。

第1.19.9節コピー

IEEE 754の値をロードして格納する（したがって、コピーする）には、FLD/FST 命令を使用しないといけなと慣性で考えるかもしれませんが。にもかかわらず、普通のMOV 命令でも同じことがより簡単に実現できます。もちろん、ビット単位で値をコピーします。

第1.19.10節スタック、計算機と逆ポーランド記法

いくつかの古い計算機が逆ポーランド記法を使用する理由を理解しました。

たとえば、12と34を追加するには12を入力し、34を入力してから「プラス」を押します。

これは、古い電卓はスタックマシンの実装なので、複雑なカッコで囲まれた式を処理するよりはるかに簡単だからです。

第1.19.11節80ビット？

FPUの内部数値表現は80ビットです。 2^n 形式ではないので、変わった数値です。おそらく歴史的な理由によるものだという仮説があります。IBMの標準的なパンチカードでは、12行の80ビットをエンコードできます。80・25 テキストモードの解像度も過去においてポピュラーでした。

ウィキペディアには別の説明があります：https://en.wikipedia.org/wiki/Extended_precision

もっと知っていたら、著者にメールを送ってください：[my emails](#)

第1.19.12節x64

浮動小数点数がx86-64でどのように処理されるかについては、これを読んでください：[1.29 on page 521](#)

第1.19.13節練習問題

- <http://challenges.re/60>
- <http://challenges.re/61>

第1.20節配列

yy¹¹⁹ 配列は、互いに隣り合って、同じ型を持つメモリ内の変数のセットです。¹²⁰

第1.20.1節単純な例

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;
```

¹¹⁹[AKA](#) 「homogener Container」.

¹²⁰[AKA](#) 「homogeneous container」

```

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};

```

x86**MSVC**

コンパイルしてみましょう。

Listing 1.221: MSVC 2008

```

_TEXT    SEGMENT
_i$ = -84                                ; size = 4
_a$ = -80                                ; size = 80
_main    PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84                      ; 00000054H
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN6@main
$LN5@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp     DWORD PTR _i$[ebp], 20      ; 00000014H
    jge     SHORT $LN4@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN5@main
$LN4@main:
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20      ; 00000014H
    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    mov     edx, DWORD PTR _a$[ebp+ecx*4]
    push    edx
    mov     eax, DWORD PTR _i$[ebp]
    push    eax
    push    OFFSET $SG2463
    call    _printf

```

```
    add     esp, 12      ; 0000000cH
    jmp     SHORT $LN2@main
$LN1@main:
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main     ENDP
```

特別なことは何もなく、2つのループだけです。1つめは配列に値を詰めるループで2つめは値を表示するループです。shl ecx, 1 命令は ECX の値を2倍するのに使用されます。詳細はこちら [1.18.2 on page 267](#)

80バイトは4バイトの20要素分の配列用としてスタック上に確保されます。

OllyDbg でこの例を試してみましょう。

配列がどのように埋まるのか見ていきます。

各要素は32ビットの *int* 型で値はインデックスを2倍したものです。

The screenshot shows the OllyDbg interface with the following components:

- Assembly View:** Displays assembly code for the CPU main thread. The instruction at address 0040102C is highlighted: `MOV DWORD PTR SS:[LOCAL.21],0`. The instruction at 0040102E is `JMP SHORT 0040103E`.
- Registers (FPU):** Shows the state of CPU registers. EIP is 0040102C, ESP is 0018FF44, and EAX is 00000000.
- Stack:** Shows the stack frame starting at 0018FEF0. The instruction `Jump from 40101C` is visible.
- Memory Dump:** Shows a hex dump of memory starting at address 0018FEF0. The dump shows a sequence of 20 elements, each 4 bytes (32 bits) in size, representing the array being filled.

図 1.87: OllyDbg: 要素を埋めた後

この配列はスタックに位置しているので、20要素すべてを見ることができます。

GCC

GCC 4.4.1ではこのようになります。

Listing 1.222: GCC 4.4.1

```

main      public main
proc near ; DATA XREF: _start+17

```

```

var_70      = dword ptr -70h
var_6C      = dword ptr -6Ch
var_68      = dword ptr -68h
i_2         = dword ptr -54h
i           = dword ptr -4

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFF0h
        sub     esp, 70h
        mov     [esp+70h+i], 0          ; i=0
        jmp     short loc_804840A

loc_80483F7:
        mov     eax, [esp+70h+i]
        mov     edx, [esp+70h+i]
        add     edx, edx                ; edx=i*2
        mov     [esp+eax*4+70h+i_2], edx
        add     [esp+70h+i], 1          ; i++

loc_804840A:
        cmp     [esp+70h+i], 13h
        jle     short loc_80483F7
        mov     [esp+70h+i], 0
        jmp     short loc_8048441

loc_804841B:
        mov     eax, [esp+70h+i]
        mov     edx, [esp+eax*4+70h+i_2]
        mov     eax, offset aADD ; "a[%d]=%d\n"
        mov     [esp+70h+var_68], edx
        mov     edx, [esp+70h+i]
        mov     [esp+70h+var_6C], edx
        mov     [esp+70h+var_70], eax
        call    _printf
        add     [esp+70h+i], 1

loc_8048441:
        cmp     [esp+70h+i], 13h
        jle     short loc_804841B
        mov     eax, 0
        leave
        retn

main      endp

```

なお、変数 *a* は *int** 型です (*int* へのポインタ) — 別の関数に配列へのポインタを渡すことができます。しかし、もっと正確には、配列の最初の要素へのポインタが渡されます。(要素の残りのアドレスは明確なやり方で計算されます)

もしこのポインタを *a[idx]* としてインデックスするなら、*idx* はポインタに加算されるだけで、配置されている要素 (計算されたポインタが示されている) がリターンされます。

面白い例: *string* のような文字列は (1) 文字の配列で *const char[]* の型を持ちます。

インデックスもこのポインタに適用されます。

そしてこれが「string」[i] のように書き込みが可能な理由です。これは C/C++ の正しい表現です！

ARM

非最適化 Keil 6/2013 (ARMモード)

	EXPORT	_main	
_main	STMFD	SP!, {R4,LR}	
	SUB	SP, SP, #0x50	; int変数20個分の場所を確保する
	; 最初のループ		
	MOV	R4, #0	; i
	B	loc_4A0	
loc_494	MOV	R0, R4, LSL#1	; R0=R4*2
	STR	R0, [SP,R4,LSL#2]	; R0にSP+R4<<2 (SP+R4*4と同様) を保存
	ADD	R4, R4, #1	; i=i+1
loc_4A0	CMP	R4, #20	; i<20?
	BLT	loc_494	; 条件を満たすなら、ループボディを再度実行する
	; 2番目のループ		
	MOV	R4, #0	; i
	B	loc_4C4	
loc_4B0	LDR	R2, [SP,R4,LSL#2]	; (printfの第二引数) R2=*(SP+R4<<4)
			(*(SP+R4*4)と同様)
	MOV	R1, R4	; (printfの第一引数) R1=i
	ADR	R0, aADD	; "a[%d]=%d\n"
	BL	__2printf	
	ADD	R4, R4, #1	; i=i+1
loc_4C4	CMP	R4, #20	; i<20?
	BLT	loc_4B0	; 条件を満たすなら、ループボディを再度実行する
	MOV	R0, #0	; 戻り値
	ADD	SP, SP, #0x50	; 確保していたint変数20個分のチャンクを開放する
	LDMFD	SP!, {R4,PC}	

int 型は32ビットのストレージを必要とします (または4バイト)。

20個の int 変数を保存するには80バイト (0x50) が必要です。だから、SUB SP, SP, #0x50 のようになっています。

関数プロローグの命令はスタックにちょうどその分の空間を確保しています。

最初と次のループの両方で、ループイテレータ i は R4 レジスタに置かれています。

配列に書かれる数は $i*2$ として計算されます。これは1ビット左シフトすることと同じで、`MOV R0, R4, LSL#1` 命令がこれを行っています。

`STR R0, [SP, R4, LSL#2]` は R0 の内容を配列に書き込んでいます。

配列の要素へのポインタがどのように計算されるかを示しています。**SP!**は配列の先頭を示しています。R4 は i です。

i を2ビット左シフトすると、4倍することに等しいです。(各配列の要素は4バイトです)そして配列の先頭アドレスに追加されます。

次のループは `LDR R2, [SP, R4, LSL#2]` 命令の逆です。配列から必要とする値をロードし、ポインタもまた同様に計算されます。

最適化 Keil 6/2013 (Thumbモード)

```

_main
    PUSH    {R4,R5,LR}
; int変数20個分+1変数の場所を確保する
    SUB     SP, SP, #0x54

; 最初のループ
    MOVS    R0, #0        ; i
    MOV     R5, SP        ; 配列要素の先頭へのポインタ

loc_1CE
    LSLS    R1, R0, #1    ; R1=i<<1 (i*2と同様)
    LSLS    R2, R0, #2    ; R2=i<<2 (i*4と同様)
    ADDS    R0, R0, #1    ; i=i+1
    CMP     R0, #20       ; i<20?
    STR     R1, [R5,R2]   ; R1を*(R5+R2)に保存 (R5+i*4と同じ)
    BLT     loc_1CE       ; 条件を満たすなら、ループボディを再度実行する

; 2番目のループ

loc_1DC
    MOVS    R4, #0        ; i=0

    LSLS    R0, R4, #2    ; R0=i<<2 (i*4と同様)
    LDR     R2, [R5,R0]   ; *(R5+R0)からロード (R5+i*4と同様)
    MOVS    R1, R4
    ADR     R0, aADD       ; "a[%d]=%d\n"
    BL      __2printf
    ADDS    R4, R4, #1    ; i=i+1
    CMP     R4, #20       ; i<20?
    BLT     loc_1DC       ; 条件を満たすなら、ループボディを再度実行する
    MOVS    R0, #0        ; 戻り値

; 確保していたint変数20個+1分のチャンクを開放する
    ADD     SP, SP, #0x54
    POP     {R4,R5,PC}

```

Thumbコードも大変似ています。

Thumbコードはビットシフト用の特別な命令を持っています (LSLS のような)。これは配列に書き込まれる値を計算し、また配列の各要素のアドレスも同様に計算します。

コンパイラはもう少し余分な空間をローカルスタックに確保します。しかし、最後の4バイトは使用されません。

非最適化 **GCC 4.9.1 (ARM64)**

Listing 1.223: 非最適化 GCC 4.9.1 (ARM64)

```
.LC0:
    .string "a[%d]=%d\n"
main:
; スタックフレームにFPとLRを保存
    stp    x29, x30, [sp, -112]!
; スタックフレームを設定 (FP=SP)
    add    x29, sp, 0
; 初期カウンタ値を0に設定 (WZRは常に0を保持するレジスタ)
    str    wzr, [x29,108]
; ループ条件チェックコードにジャンプ
    b      .L2
.L3:
; "i" 変数の値をロード
    ldr    w0, [x29,108]
; 2倍する
    lsl    w2, w0, 1
; ローカルスタックの配列の位置を見つける
    add    x0, x29, 24
; ローカルスタックから符号なし32ビットをロードし、64ビットの符号付き値へ拡張
    ldrsw  x1, [x29,108]
; 要素のアドレスを計算 (X0+X1<<2=array address+i*4) し、そこにW2 (i*2) を保存
    str    w2, [x0,x1,lsl 2]
; カウンタ (i) をインクリメント
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L2:
; 終了かをチェック
    ldr    w0, [x29,108]
    cmp    w0, 19
; 終了でなければL3にジャンプ (ループボディの開始)
    ble    .L3
; 関数の第二部がここから始まる
; 初期カウンタ変数を0に設定
; ところで、ローカルスタックの同じ場所がカウンタとして使用されます
; 同じローカル変数 (i) がカウンタとして使用されるからです
    str    wzr, [x29,108]
    b      .L4
.L5:
; 配列のアドレスを計算
    add    x0, x29, 24
; "i" 値をロード
    ldrsw  x1, [x29,108]
```

```

; アドレス (X0+X1<<2 = address of array + i*4) の配列から値をロード
    ldr    w2, [x0,x1,ls1 2]
; "a[%d]=%d\n" 文字列のアドレスをロード
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
; "i" 変数をW1にロードし、printf() の第二引数として渡す
    ldr    w1, [x29,108]
; W2はロードされた配列要素の値を保持
; printf() を呼び出す
    bl     printf
; "i" 変数をインクリメント
    ldr    w0, [x29,108]
    add    w0, w0, 1
    str    w0, [x29,108]
.L4:
; 終了した?
    ldr    w0, [x29,108]
    cmp    w0, 19
; 終了していなければループボディの開始にジャンプ
    ble    .L5
; 0をリターン
    mov    w0, 0
; FPとLRを戻す
    ldp    x29, x30, [sp], 112
    ret

```

MIPS

関数は保存しなくてはならないたくさんの S- レジスタを使用します。よって、値は関数プロローグで保存され、エピローグでリストアされます。

Listing 1.224: 最適化 GCC 4.4.5 (IDA)

```

main:
var_70      = -0x70
var_68      = -0x68
var_14      = -0x14
var_10      = -0x10
var_C       = -0xC
var_8       = -8
var_4       = -4
; 関数プロローグ
    lui    $gp, (__gnu_local_gp >> 16)
    addiu  $sp, -0x80
    la     $gp, (__gnu_local_gp & 0xFFFF)
    sw     $ra, 0x80+var_4($sp)
    sw     $s3, 0x80+var_8($sp)
    sw     $s2, 0x80+var_C($sp)
    sw     $s1, 0x80+var_10($sp)
    sw     $s0, 0x80+var_14($sp)
    sw     $gp, 0x80+var_70($sp)
    addiu  $s1, $sp, 0x80+var_68

```

```

        move    $v1, $s1
        move    $v0, $zero
; この値はループ終端として使用されます
; コンパイル時にGCCコンパイラによって事前に計算されます
        li      $a0, 0x28 # '('

loc_34:                                     # CODE XREF: main+3C
; 値をメモリに保存
        sw      $v0, 0($v1)
; 各イテレーション毎に値を2インクリメント
        addiu   $v0, 2
; ループ終端に到達した?
        bne     $v0, $a0, loc_34
; アドレスを4増やす
        addiu   $v1, 4
; 配列に詰め込むループが終了
; 次のループを開始
        la      $s3, $LC0                # "a[%d]=%d\n"
; "i" 変数が $s0に存在
        move    $s0, $zero
        li      $s2, 0x14

loc_54:                                     # CODE XREF: main+70
; printf() を呼び出す
        lw      $t9, (printf & 0xFFFF)($gp)
        lw      $a2, 0($s1)
        move    $a1, $s0
        move    $a0, $s3
        jalr    $t9
; "i" をインクリメント
        addiu   $s0, 1
        lw      $gp, 0x80+var_70($sp)
; ループの終了に到達していなければループボディにジャンプ
        bne     $s0, $s2, loc_54
; メモリポインタを次の32ビットwordに移動する
        addiu   $s1, 4
; 関数エピローグ
        lw      $ra, 0x80+var_4($sp)
        move    $v0, $zero
        lw      $s3, 0x80+var_8($sp)
        lw      $s2, 0x80+var_C($sp)
        lw      $s1, 0x80+var_10($sp)
        lw      $s0, 0x80+var_14($sp)
        jr      $ra
        addiu   $sp, 0x80

$LC0:      .ascii "a[%d]=%d\n"<0>      # DATA XREF: main+44

```

面白いこと：2つのループがあり、最初のループは i がいきりません。 $i * 2$ が必要なだけです（各イテレーションで2をインクリメントする）。それとメモリ上のアドレスが必要です（各イテレーションで4を増やす）。

だから、2つの変数を確認します。1つは（\$V0）毎回2を増やし、もう1つは4増やします

(\$V1)。

次のループは `printf()` が呼び出されるところです。*i* の値をユーザに報告します。毎回1増やす変数があり (\$S0)、そしてメモリアドレス (\$S1) も毎回4増えます。

前に検討したループ最適化を私たちに思いださせます：?? on page ??

目的は乗算を取り除くことです。

第1.20.2節バッファオーバーフロー

配列の範囲外の読み込み

配列のインデックス化は単に `array[index]` です。生成されたコードを詳しく研究したなら、20未満であるかチェックするようなインデックスの境界チェックがないことに気づくでしょう。もしインデックスが20以上だったらどうでしょうか。これは C/C++ が批判される1つの特徴です。

コンパイルされて動作するコードがあります。

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    printf ("a[20]=%d\n", a[20]);

    return 0;
};
```

コンパイル結果 (MSVC 2008)

Listing 1.225: 非最適化 MSVC 2008

```
$SG2474 DB      'a[20]=%d', 0aH, 00H

_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 84
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN3@main
$LN2@main:
    mov     eax, DWORD PTR _i$[ebp]
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp     DWORD PTR _i$[ebp], 20
```



```

    jge     SHORT $LN1@main
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN2@main
$LN1@main:
    mov     eax, DWORD PTR _a$[ebp+80]
    push    eax
    push    OFFSET $SG2474 ; 'a[20]=%d'
    call    DWORD PTR __imp__printf
    add     esp, 8
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main      ENDP
_TEXT      ENDS
END

```

コードは次の結果を生成します。

Listing 1.226: OllyDbg: console output

```
a[20]=1638280
```

これは単に配列のそばのスタックにある 何かです。配列の最初の要素から80バイト離れています。

この値がどこから来るのか OllyDbg を使って見つけてみましょう。

最後の配列の要素のすぐあとに配置された値をロードして見つけましょう。

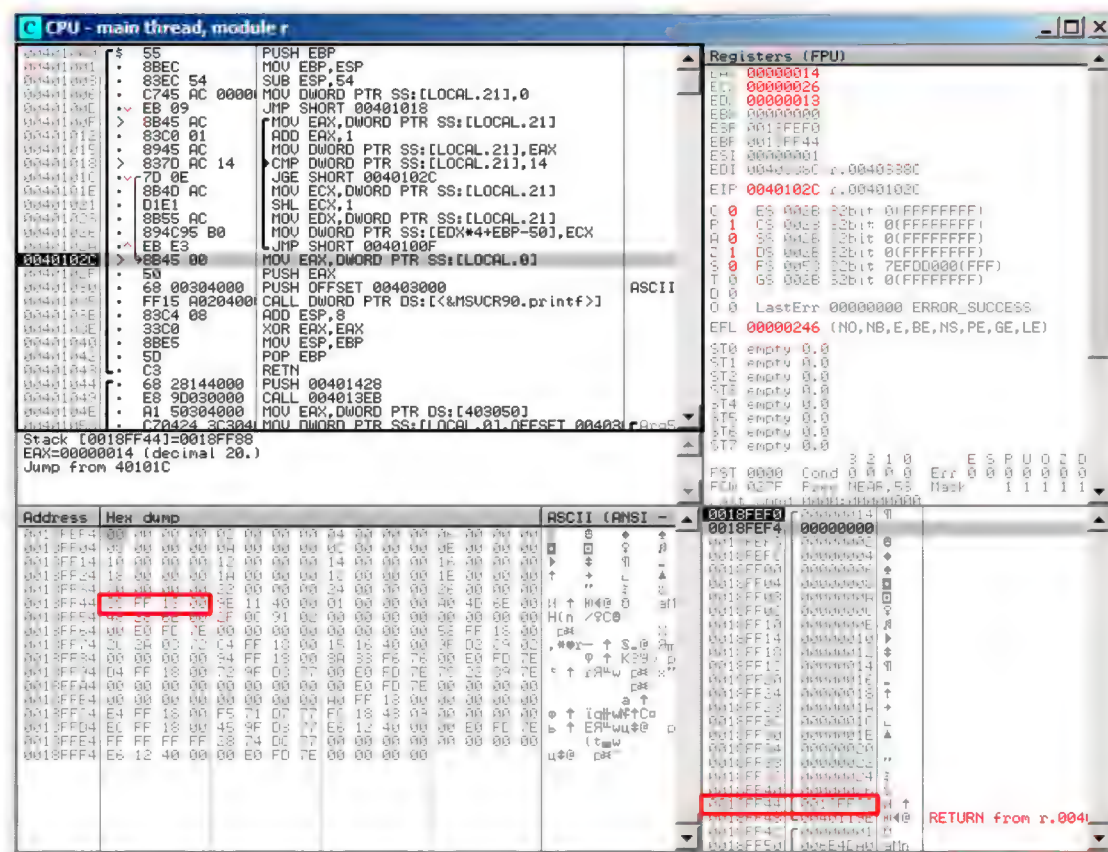


図 1.88: OllyDbg: 20番目の要素を読み込み、printf() を実行する

これは何でしょうか？スタックレイアウトで判断すると、これは保存されたEBPレジスタの値です。

もっとトレースしてどのようにリストアされるか見てみましょう。

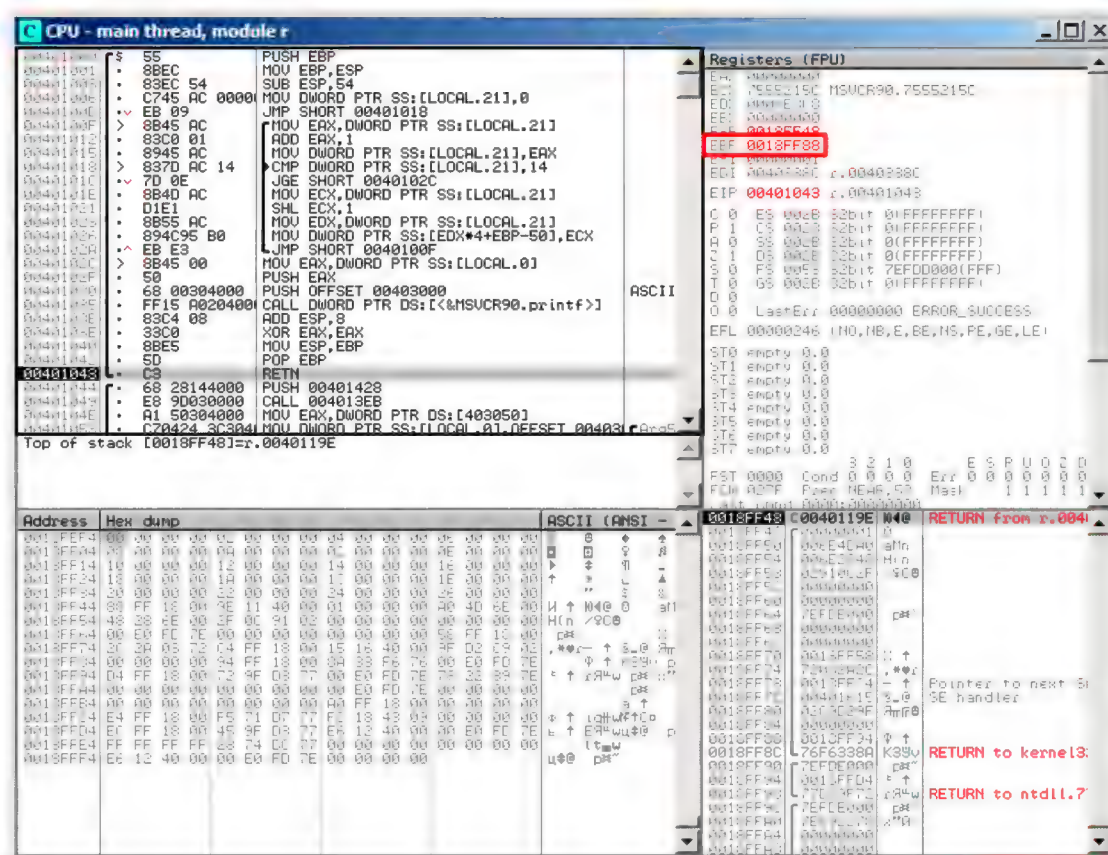


図 1.89: OllyDbg: EBPの値をリストア

本当に、異なっていますか？コンパイラはインデックス値が配列の境界内かを常にチェックする追加のコードを生成するかもしれません。（高水準プログラミング言語¹²¹のように）しかし、これはコードを遅くします。

配列境界を越えて書きこむ

私たちはスタックからいくつかの値を不正に読んでいますが、何かを書くことができたかどうか？

こういう風になります。

```
#include <stdio.h>
```

```
int main()
{
```

¹²¹Java, Pythonなど

```

    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
};

```

MSVC

そしてこうなります。

Listing 1.227: 非最適化 MSVC 2008

```

_TEXT    SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80
_main    PROC
push     ebp
mov      ebp, esp
sub      esp, 84
mov      DWORD PTR _i$[ebp], 0
jmp      SHORT $LN3@main
$LN2@main:
mov      eax, DWORD PTR _i$[ebp]
add      eax, 1
mov      DWORD PTR _i$[ebp], eax
$LN3@main:
cmp      DWORD PTR _i$[ebp], 30 ; 0000001eH
jge      SHORT $LN1@main
mov      ecx, DWORD PTR _i$[ebp]
mov      edx, DWORD PTR _i$[ebp] ; この命令は明らかに冗長
mov      DWORD PTR _a$[ebp+ecx*4], edx ; ECXは2つめのオペランドとして使用できます
jmp      SHORT $LN2@main
$LN1@main:
xor      eax, eax
mov      esp, ebp
pop      ebp
ret      0
_main    ENDP

```

コンパイルしたプログラムは起動後にクラッシュします。当然です。どこでクラッシュするか正確にみてみましょう。

OllyDbg でロードし、30要素が書かれるまでトレースしてみましょう。

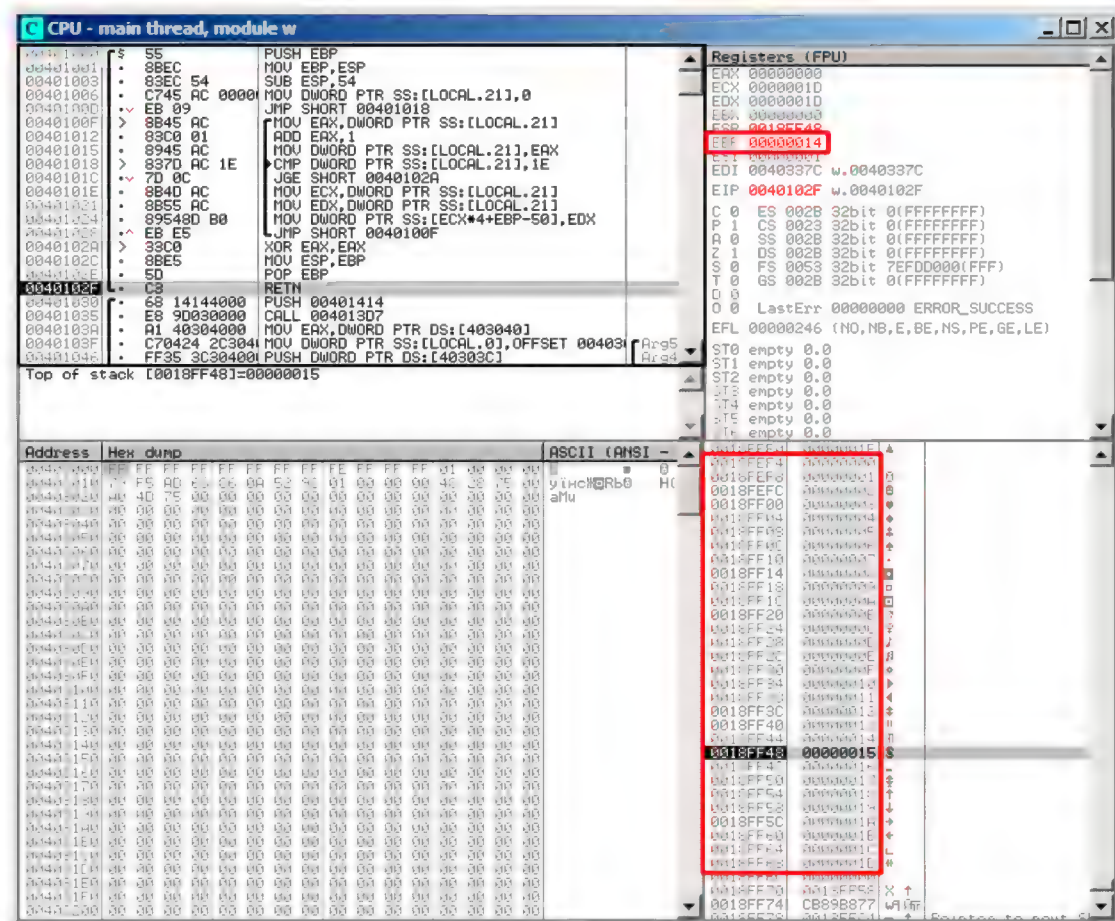


図 1.90: OllyDbg: EBPの値をリストアした後

関数が終了するまでトレースします。

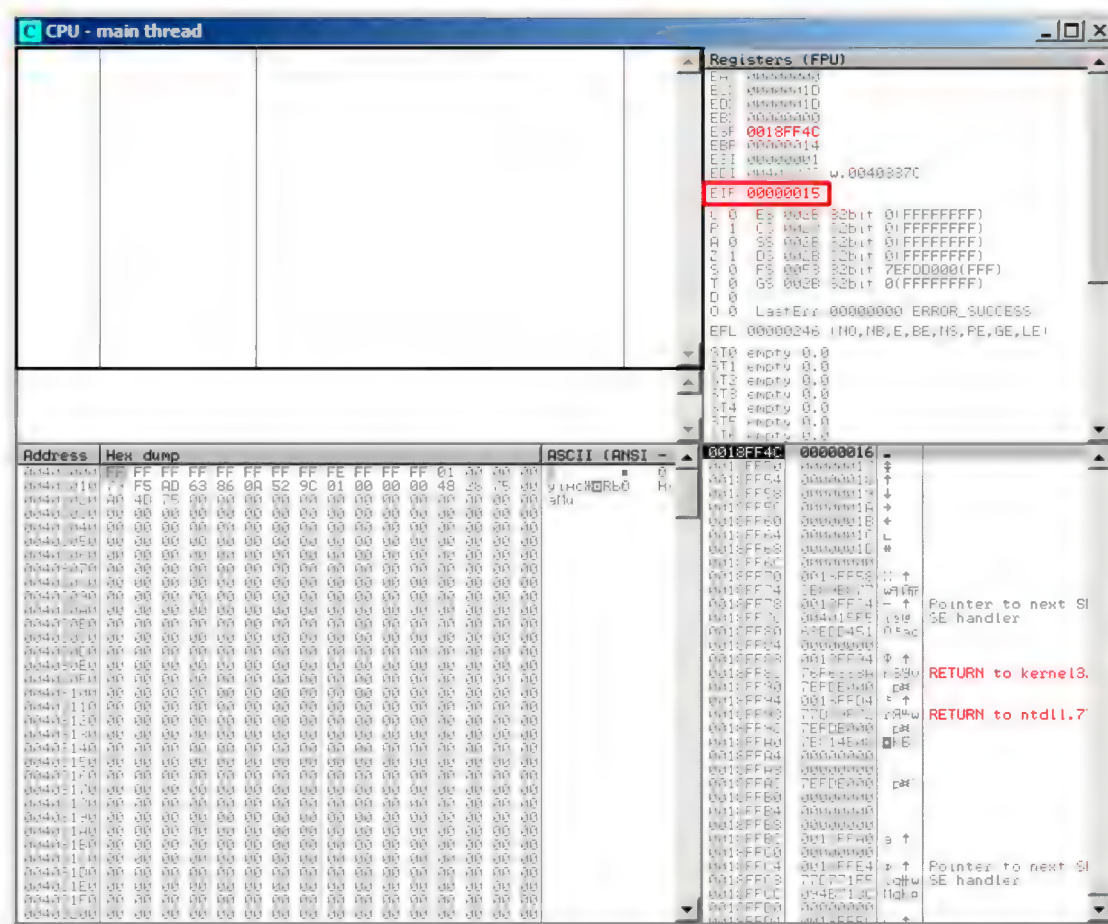


図 1.91: OllyDbg: EIP がリストアされるが、OllyDbg は0x15でディスアセンブルできない

レジスタをよく見てください。

EIP は0x15です。コードでは不正なアドレスではありません。少なくともwin32のコードとしては！我々の意志に反しています。EBP レジスタが0x14を、ECX と EDX が0x1Dを含んでいるということが面白いです。

スタックレイアウトをもう少し勉強しましょう。

制御フローが `main()` を通ったあと、EBP レジスタの値はスタックに保存されます。それから、84バイトが配列と `i` 用に確保されます。それは $(20+1)*\text{sizeof}(\text{int})$ です。ESP はローカルスタックの `_i` 変数を指し、次の `PUSH something` の実行の後で、何かが次の `_i` に現れます。

これが、制御が `main()` にあるときのスタックレイアウトです。

ESP	4バイトが <i>i</i> 変数に確保される
ESP+4	80バイトが a[20] 配列に確保される
ESP+84	EBP の値を保存
ESP+88	リターンアドレス

a[19]=something 文は配列の境界である最後の *int* を書き込みます (今は境界内です!)。

a[20]=something 文は EBP の値が保存された場所に 何かを書き込みます。

クラッシュ時のレジスタの状態を見てください。我々の場合、20番目の要素に20が書かれています。関数の最後で、関数エピローグがオリジナルの EBP 値をリストアします。(10進数の20は16進数で 0x14 です)。そして、RET が実行されます。これは POP EIP 命令と同じ効果です。

RET 命令はスタックからリターンアドレスを取って(これはCRTの中のアドレスで、main() を呼び出したアドレスです)、21が保存されます (16進数で 0x15)。CPUはアドレス 0x15 をトラップしますが、実行可能なコードがここにはないので、例外が発生します。

ようこそ！バッファオーバーフローです。¹²²

int 配列を文字列 (*char* 配列) で置換するには、意図的に長い文字列を作成し、それをプログラムに渡し、関数に渡し、文字列の長さをチェックせず、より短いバッファにコピーし、そこにジャンプするアドレスをプログラムに指し示すことで可能になります。実際にはそんなに簡単ではありませんが、それが現実にどのように現れたかが重要です。古典的な記事は : [Aleph One, *Smashing The Stack For Fun And Profit*, (1996)]¹²³

GCC

GCC 4.4.1 で同じコードを試してみましょう。次を得ます。

```

main                public main
                   proc near

a                   = dword ptr -54h
i                   = dword ptr -4

                   push    ebp
                   mov     ebp, esp
                   sub     esp, 60h ; 96
                   mov     [ebp+i], 0
                   jmp     short loc_80483D1

loc_80483C3:        mov     eax, [ebp+i]
                   mov     edx, [ebp+i]
                   mov     [ebp+eax*4+a], edx
                   add     [ebp+i], 1

loc_80483D1:        cmp     [ebp+i], 1Dh
                   jle     short loc_80483C3
                   mov     eax, 0
                   leave

```

¹²² [wikipedia](#)

¹²³ 以下で利用可能 <http://yurichev.com/mirrors/phrack/p49-0x0e.txt>

```

main                retn
                    endp

```

Linuxで動かすと Segmentation fault が発生します。

GDBデバッガで動かすと、このようになります。

```

(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax                0x0          0
ecx                0xd2f96388    -755407992
edx                0x1d          29
ebx                0x26eff4      2551796
esp                0xbffff4b0    0xbffff4b0
ebp                0x15          0x15
esi                0x0          0
edi                0x0          0
eip                0x16          0x16
eflags             0x10202      [ IF RF ]
cs                 0x73          115
ss                 0x7b          123
ds                 0x7b          123
es                 0x7b          123
fs                 0x0          0
gs                 0x33          51
(gdb)

```

レジスタ値はwin32の例とは少し異なりますし、スタックレイアウトも少し違います。

第1.20.3節バッファオーバーフロー保護手法

このソースコードに対する保護手法はいくつかあり、C/C++ プログラムの怠慢にもかかわらず、MSVCにはオプションがあります。¹²⁴

```

/RTCs スタックフレームの実行時チェック
/GZ スタックチェックの有効化 (/RTCs)

```

手法の1つに関数プロローグでスタックのローカル変数の間にランダムな値を書き込み、関数を終了する前に関数エピローグでそれをチェックするというものがあります。値が同じでなければ、最後の命令 RET を実行せず、停止（ハング）します。プロセスは停止しますが、遠隔の攻撃者があなたのホストを攻撃するよりはよいことです。

このランダムな値はしばしば「カナリア」と呼ばれ、炭鉱労働でのカナリアに関連しています。¹²⁵ 昔、有毒なガスを一早く検知できるよう、炭鉱労働者に使用されていました。

¹²⁴ コンパイラサイドのバッファオーバーフロー保護手法: wikipedia.org/wiki/Buffer_overflow_protection

¹²⁵ wikipedia.org/wiki/Domestic_canary#Miner.27s_canary

カナリアは炭鉱のガスにとっても敏感で、危機の際に騒ぎ立て、場合によっては死んでしまいました。

とてもシンプルな配列の例を [MSVC](#) でRTC1とRTCsオプション付きでコンパイルする場合 ([1.20.1 on page 322](#)),「カナリア」が正しいかどうか、関数の最後に `@_RTC_CheckStackVars@8` を呼び出すのを見ることができます。

GCCがこれをどのように扱うかを見てみましょう。 `alloca()` ([1.7.2 on page 45](#)) の例を扱しましょう。

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

デフォルトでは、追加のオプションなしに、GCC 4.7.3は「カナリア」チェックをコードに挿入します。

Listing 1.228: GCC 4.7.3

```
.LC0:
    .string "hi! %d, %d, %d\n"
f:
    push    ebp
    mov     ebp, esp
    push    ebx
    sub     esp, 676
    lea     ebx, [esp+39]
    and     ebx, -16
    mov     DWORD PTR [esp+20], 3
    mov     DWORD PTR [esp+16], 2
    mov     DWORD PTR [esp+12], 1
    mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov     DWORD PTR [esp+4], 600
    mov     DWORD PTR [esp], ebx
    mov     eax, DWORD PTR gs:20 ; スタックカナリア
    mov     DWORD PTR [ebp-12], eax
    xor     eax, eax
    call    _snprintf
    mov     DWORD PTR [esp], ebx
```

```

    call    puts
    mov     eax, DWORD PTR [ebp-12]
    xor     eax, DWORD PTR gs:20      ; カナリアをチェック
    jne     .L5
    mov     ebx, DWORD PTR [ebp-4]
    leave
    ret
.L5:
    call    __stack_chk_fail

```

ランダム値が `gs:20` に配置されます。スタックに書かれて、関数の最後でスタックの値が `gs:20` の「カナリア」と一致しているか比較します。値が一致していなければ、`__stack_chk_fail` 関数が呼び出され、ときどき (Ubuntu 13.04 x86): のようなものをコンソールでみることがあります。

```

*** buffer overflow detected ***: ./2_1 terminated
===== Backtrace: =====
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2_1
0804a000-0804b000 rw-p 00001000 08:01 2097586 /home/dennis/2_1
094d1000-094f2000 rw-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-linux-gnu/
↳ libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602 /lib/i386-linux-gnu/
↳ libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602 /lib/i386-linux-gnu/
↳ libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-linux-gnu/libc
↳ -2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781 /lib/i386-linux-gnu/libc
↳ -2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781 /lib/i386-linux-gnu/libc
↳ -2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0 [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794 /lib/i386-linux-gnu/ld
↳ -2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794 /lib/i386-linux-gnu/ld
↳ -2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794 /lib/i386-linux-gnu/ld
↳ -2.17.so

```

```

bff35000-bff56000 rw-p 00000000 00:00 0          [stack]
Aborted (core dumped)

```

gsはいわゆるセグメントレジスタです。このレジスタは広くMS-DOSやDOS拡張で使用されました。今日、この機能は異なっています。

簡単に言うと、Linuxでの gs レジスタは常にTLS¹²⁶ (?? on page ??) を指し示します。スレッド固有の情報がそこに保存されます。ところで、win32では fs レジスタは同じ役割を担い、TIB¹²⁷を指し示します。¹²⁸

より詳細はLinuxカーネルソースコード *arch/x86/include/asm/stackprotector.h* の中にコメントとして記述してあるのを見つけられます (少なくとも3.11バージョンには)。

最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

単純な配列の例に戻りましょう (1.20.1 on page 322)。

繰り返しますが、LLVMが「カナリア」の正しさをどのようにチェックするのか見ることができます。

```

_main
var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
var_1C          = -0x1C
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

    PUSH        {R4-R7,LR}
    ADD         R7, SP, #0xC
    STR.W       R8, [SP,#0xC+var_10]!
    SUB         SP, SP, #0x54
    MOVW        R0, #a0bjc_methtype ; "objc_methtype"

```

¹²⁶Thread Local Storage

¹²⁷Thread Information Block

¹²⁸wikipedia.org/wiki/Win32_Thread_Information_Block

```

MOVS    R2, #0
MOVT.W  R0, #0
MOVS    R5, #0
ADD     R0, PC
LDR.W   R8, [R0]
LDR.W   R0, [R8]
STR     R0, [SP,#0x64+canary]
MOVS    R0, #2
STR     R2, [SP,#0x64+var_64]
STR     R0, [SP,#0x64+var_60]
MOVS    R0, #4
STR     R0, [SP,#0x64+var_5C]
MOVS    R0, #6
STR     R0, [SP,#0x64+var_58]
MOVS    R0, #8
STR     R0, [SP,#0x64+var_54]
MOVS    R0, #0xA
STR     R0, [SP,#0x64+var_50]
MOVS    R0, #0xC
STR     R0, [SP,#0x64+var_4C]
MOVS    R0, #0xE
STR     R0, [SP,#0x64+var_48]
MOVS    R0, #0x10
STR     R0, [SP,#0x64+var_44]
MOVS    R0, #0x12
STR     R0, [SP,#0x64+var_40]
MOVS    R0, #0x14
STR     R0, [SP,#0x64+var_3C]
MOVS    R0, #0x16
STR     R0, [SP,#0x64+var_38]
MOVS    R0, #0x18
STR     R0, [SP,#0x64+var_34]
MOVS    R0, #0x1A
STR     R0, [SP,#0x64+var_30]
MOVS    R0, #0x1C
STR     R0, [SP,#0x64+var_2C]
MOVS    R0, #0x1E
STR     R0, [SP,#0x64+var_28]
MOVS    R0, #0x20
STR     R0, [SP,#0x64+var_24]
MOVS    R0, #0x22
STR     R0, [SP,#0x64+var_20]
MOVS    R0, #0x24
STR     R0, [SP,#0x64+var_1C]
MOVS    R0, #0x26
STR     R0, [SP,#0x64+var_18]
MOV     R4, 0xFDA ; "a[%d]=%d\n"
MOV     R0, SP
ADDS    R6, R0, #4
ADD     R4, PC
B       loc_2F1C

```

; 次のループを繰り返す

```

loc_2F14
    ADDS     R0, R5, #1
    LDR.W    R2, [R6,R5,LSL#2]
    MOV      R5, R0

loc_2F1C
    MOV      R0, R4
    MOV      R1, R5
    BLX      _printf
    CMP      R5, #0x13
    BNE      loc_2F14
    LDR.W    R0, [R8]
    LDR      R1, [SP,#0x64+canary]
    CMP      R0, R1
    ITTTT EQ ; カナリアは正しいか?
    MOVEQ    R0, #0
    ADDEQ    SP, SP, #0x54
    LDREQ.W  R8, [SP+0x64+var_64],#4
    POPEQ    {R4-R7,PC}
    BLX      ___stack_chk_fail

```

まず最初に、見てきたように、LLVMはループを「展開し」、LLVMは高速になると結論づけて、事前に計算されて値はすべて配列に1つ1つ書かれます。なお、ARMモードでの命令はこれをより高速にする手助けをするかもしれません。これを見つけるのは宿題にします。

関数の最後で「カナリア」の比較を見ます。ローカルスタックのカナリアと R8 で指し示した正しいものの。

それぞれが一致していれば、4命令ブロックが ITTTT EQ で実行され、R0 に0が書かれ、関数エピローグが終了します。「カナリア」が一致していなければ、ブロックがスキップされ、___stack_chk_fail 関数へのジャンプが実行され、おそらく実行が停止されます。

第1.20.4節配列についてもう少し

今や C/C++ のコードでこのように書き込むのが不可能なことを理解しています。

```

void f(int size)
{
    int a[size];
    ...
};

```

コンパイラはコンパイル時にローカルスタックレイアウト上の場所を確保するために正確な配列のサイズを知る必要があります。

配列の任意のサイズを必要とする場合、malloc() を使用して確保し、そして確保したメモリブロックに必要とする型の変数の配列としてアクセスします。

またはC99標準の機能 [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.5/2] を使用します。内部で alloca() (1.7.2 on page 45) を使用しているかのように働きます。

C用のガーベッジコレクションライブラリを使用することも可能です。

C++向けにスマートポインタをサポートするライブラリもあります。

第1.20.5節文字列へのポインタの配列

ここでは、ポインタの配列の例を示します。

Listing 1.229: Get month name

```
#include <stdio.h>

const char* month1[]=
{
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
};

// 0~11の範囲で
const char* get_month1 (int month)
{
    return month1[month];
};
```

x64

Listing 1.230: 最適化 MSVC 2013 x64

DATA	SEGMENT
month1	DQ FLAT:\$SG3122
	DQ FLAT:\$SG3123
	DQ FLAT:\$SG3124
	DQ FLAT:\$SG3125
	DQ FLAT:\$SG3126
	DQ FLAT:\$SG3127
	DQ FLAT:\$SG3128
	DQ FLAT:\$SG3129
	DQ FLAT:\$SG3130
	DQ FLAT:\$SG3131
	DQ FLAT:\$SG3132
	DQ FLAT:\$SG3133
\$SG3122	DB 'January', 00H
\$SG3123	DB 'February', 00H
\$SG3124	DB 'March', 00H
\$SG3125	DB 'April', 00H
\$SG3126	DB 'May', 00H
\$SG3127	DB 'June', 00H
\$SG3128	DB 'July', 00H
\$SG3129	DB 'August', 00H
\$SG3130	DB 'September', 00H
\$SG3156	DB '%s', 0aH, 00H
\$SG3131	DB 'October', 00H
\$SG3132	DB 'November', 00H
\$SG3133	DB 'December', 00H

```

_DATA    ENDS

month$ = 8
get_month1 PROC
    movsxd    rax, ecx
    lea       rcx, OFFSET FLAT:month1
    mov       rax, QWORD PTR [rcx+rax*8]
    ret       0
get_month1 ENDP

```

コードはとても単純です。

- 最初の MOVSDX 命令は、ECX (*month* 引数が渡される) から32ビットの値を符号拡張付きの RAX (*month* 引数は *int* 型なので) にコピーします。

符号拡張の理由は、この32ビット値が他の64ビット値との計算に使用されるためです。

したがって、64ビット142に昇格させる必要があります。¹²⁹

- 次にポインタテーブルのアドレスが RCX にロードされます。
- 最後に、入力値 (*month*) に8を掛けてアドレスに加算します。確かに：私たちは64ビット環境にあり、すべてのアドレス (またはポインタ) は正確に64ビット (または8バイト) の記憶域を必要とします。したがって、各テーブル要素は8バイト幅です。それで、なぜ特定の要素 $month * 8$ をスキップする必要があるのでしょうか。これが MOV が行うことです。さらに、この命令はこのアドレスの要素もロードします。1の場合、要素は「February」などを含む文字列へのポインタになります。

最適化 GCC 4.9はもっとよく仕事をこなします。¹³⁰

Listing 1.231: 最適化 GCC 4.9 x64

```

movsx    rdi, edi
mov      rax, QWORD PTR month1[0+rdi*8]
ret

```

32ビットMSVC

32ビットMSVCコンパイラでもコンパイルしてみましょう。

Listing 1.232: 最適化 MSVC 2013 x86

```

_month$ = 8
_get_month1 PROC
    mov     eax, DWORD PTR _month$[esp-4]
    mov     eax, DWORD PTR _month1[eax*4]
    ret     0
_get_month1 ENDP

```

¹²⁹ やや奇妙ですが、負の配列インデックスはここで *month* として渡すことができます (負の配列インデックスは後で説明します: ??)。これが起こると、*int* 型の負の入力値が正しく符号拡張され、テーブルの前の対応する要素が選択されます。符号拡張なしでは正しく動作しません。

¹³⁰ GCCアセンブラ出力が排除するのに十分なほど整っていないので、「0+」がリストに残っていました。それは変位であり、ここではゼロです。

入力値は64ビットに拡張する必要がないので、そのまま使われます。

そして4倍されます。テーブル要素が32ビット（または4バイト）幅だからです。

32ビット ARM

ARMモードでのARM

Listing 1.233: 最適化 Keil 6/2013 (ARMモード)

```
get_month1 PROC
    LDR    r1, |L0.100|
    LDR    r0, [r1, r0, LSL #2]
    BX     lr
ENDP

|L0.100|
    DCD    ||.data||

    DCB    "January", 0
    DCB    "February", 0
    DCB    "March", 0
    DCB    "April", 0
    DCB    "May", 0
    DCB    "June", 0
    DCB    "July", 0
    DCB    "August", 0
    DCB    "September", 0
    DCB    "October", 0
    DCB    "November", 0
    DCB    "December", 0

    AREA  ||.data||, DATA, ALIGN=2
month1
    DCD    ||.conststring||
    DCD    ||.conststring||+0x8
    DCD    ||.conststring||+0x11
    DCD    ||.conststring||+0x17
    DCD    ||.conststring||+0x1d
    DCD    ||.conststring||+0x21
    DCD    ||.conststring||+0x26
    DCD    ||.conststring||+0x2b
    DCD    ||.conststring||+0x32
    DCD    ||.conststring||+0x3c
    DCD    ||.conststring||+0x44
    DCD    ||.conststring||+0x4d
```

テーブルのアドレスはR1にロードされます。

残りのすべては LDR 命令1つだけを使って行われます。

入力値 *month* は2ビット左シフトします（4倍するのと同じです）。それから R1に加えます（テーブルのアドレスの場所）。そしてテーブル要素はこのアドレスからロードされ

ます。

32ビットテーブル要素はテーブルからR0にロードされます。

ThumbモードでのARM

コードはほとんど同じですが、より密度が低いです。LSL サフィックスは LDR 命令では特定できないからです。

```
get_month1 PROC
    LSLS    r0,r0,#2
    LDR     r1,|L0.64|
    LDR     r0,[r1,r0]
    BX     lr
ENDP
```

ARM64

Listing 1.234: 最適化 GCC 4.9 ARM64

```
get_month1:
    adrp    x1, .LANCHOR0
    add     x1, x1, :lo12:.LANCHOR0
    ldr     x0, [x1,w0,sxtw 3]
    ret

.LANCHOR0 = . + 0
.type     month1, %object
.size     month1, 96
month1:
    .xword  .LC2
    .xword  .LC3
    .xword  .LC4
    .xword  .LC5
    .xword  .LC6
    .xword  .LC7
    .xword  .LC8
    .xword  .LC9
    .xword  .LC10
    .xword  .LC11
    .xword  .LC12
    .xword  .LC13

.LC2:
    .string "January"
.LC3:
    .string "February"
.LC4:
    .string "March"
.LC5:
    .string "April"
.LC6:
    .string "May"
```

```

.LC7:
.string "June"
.LC8:
.string "July"
.LC9:
.string "August"
.LC10:
.string "September"
.LC11:
.string "October"
.LC12:
.string "November"
.LC13:
.string "December"

```

テーブルのアドレスは ADRP/ADD 命令の組を使ってX1にロードされます。

それから付随する要素 LDR を使って選ばれて、W0を取ります（入力引数 *month* の場所のレジスタ）。左に3ビットシフトします（8倍するのと同じです）。符号拡張し（「sxtw」サフィックスが暗示しています）、X0に加算します。それから64ビット値がテーブルからX0にロードされます。

MIPS

Listing 1.235: 最適化 GCC 4.4.5 (IDA)

```

get_month1:
; テーブルのアドレスを $v0にロード
    la      $v0, month1
; 入力値を4倍する
    sll     $a0, 2
; テーブルのアドレスと掛け合わされた値を合計する
    addu    $a0, $v0
; このアドレスのテーブルの要素を $v0にロードする
    lw      $v0, 0($a0)
; リターン
    jr      $ra
    or      $at, $zero ; branch delay slot, NOP

    .data # .data.rel.local
    .globl month1
month1:
    .word aJanuary      # "1月"
    .word aFebruary     # "2月"
    .word aMarch        # "3月"
    .word aApril        # "4月"
    .word aMay          # "5月"
    .word aJune         # "6月"
    .word aJuly         # "7月"
    .word aAugust       # "8月"
    .word aSeptember   # "9月"
    .word aOctober      # "10月"
    .word aNovember    # "11月"
    .word aDecember     # "12月"

```

```

.data # .rodata.str1.4
aJanuary: .ascii "January"<0>
aFebruary: .ascii "February"<0>
aMarch: .ascii "March"<0>
aApril: .ascii "April"<0>
aMay: .ascii "May"<0>
aJune: .ascii "June"<0>
aJuly: .ascii "July"<0>
aAugust: .ascii "August"<0>
aSeptember: .ascii "September"<0>
aOctober: .ascii "October"<0>
aNovember: .ascii "November"<0>
aDecember: .ascii "December"<0>

```

配列オーバーフロー

関数は0～11の範囲の値を受け付けますが、12は通すでしょうか？テーブルにはその場所の要素はありません。

なので関数はそこにたまたまある値をロードしてリターンします。

すぐ後で、他の関数がこのアドレスからテキスト文字列を取得しようとしてクラッシュするかもしれません。

例をwin64用としてMSVCでコンパイルして、テーブルの後にリンカーが何を配置したのかを [IDA](#) で見てみましょう。

Listing 1.236: IDAでの実行可能ファイル

```

off_140011000  dq offset aJanuary_1      ; DATA XREF: .text:00000000140001003
                                     ; "January"
                                     dq offset aFebruary_1      ; "February"
                                     dq offset aMarch_1        ; "March"
                                     dq offset aApril_1         ; "April"
                                     dq offset aMay_1           ; "May"
                                     dq offset aJune_1          ; "June"
                                     dq offset aJuly_1           ; "July"
                                     dq offset aAugust_1        ; "August"
                                     dq offset aSeptember_1     ; "September"
                                     dq offset aOctober_1       ; "October"
                                     dq offset aNovember_1      ; "November"
                                     dq offset aDecember_1      ; "December"
aJanuary_1     db 'January',0      ; DATA XREF: sub_140001020+4
                                     ; .data:off_140011000
aFebruary_1    db 'February',0     ; DATA XREF: .data:00000000140011008
                                     align 4
aMarch_1       db 'March',0        ; DATA XREF: .data:00000000140011010
                                     align 4
aApril_1       db 'April',0        ; DATA XREF: .data:00000000140011018

```

月の名前がそのあとに來ています。

プログラムは小さいので、データセグメントにパックされるデータは多くありません。だから単に次の名前が来ています。しかし注意すべきはリンカーが配置するように決定するのはどんなものもありえます。

だからもし12が関数に渡されたら？13番目の要素がリターンされます。

CPUがそこにあるバイトを64ビットの値としてどのように扱うかをみてみましょう。

Listing 1.237: IDAでの実行可能ファイル

```

off_140011000    dq offset qword_140011060
                  ; DATA XREF: .text:00000000140001003
                  dq offset aFebruary_1    ; "February"
                  dq offset aMarch_1       ; "March"
                  dq offset aApril_1       ; "April"
                  dq offset aMay_1         ; "May"
                  dq offset aJune_1        ; "June"
                  dq offset aJuly_1        ; "July"
                  dq offset aAugust_1      ; "August"
                  dq offset aSeptember_1   ; "September"
                  dq offset aOctober_1     ; "October"
                  dq offset aNovember_1    ; "November"
                  dq offset aDecember_1    ; "December"
qword_140011060  dq 797261756E614Ah        ; DATA XREF: sub_140001020+4
                  ; .data:off_140011000
aFebruary_1     db 'February',0           ; DATA XREF: .data:00000000140011008
                  align 4
aMarch_1        db 'March',0             ; DATA XREF: .data:00000000140011010

```

0x797261756E614Aです。

すぐ後で、他の関数（おそらく文字列を扱う関数）がこのアドレスでバイトを読み込もうとすると、C言語の文字列を期待します。

十中八九、クラッシュします。この値は有効なアドレスのようには見えないからです。

配列オーバーフロー保護

失敗する可能性のあるものは、失敗する。

マーフィーの法則

あなたの関数を使用するプログラマはみな11より大きな値を引数として渡さないと期待するのはちょっとナイーブです。

問題をできるだけ早く報告し停止することを意味する「fail early and fail loudly」または「早く失敗する」という哲学があります。

そのような方法の1つに C/C++ のassertionがあります。

不正な値が通ってきたら、失敗するようにプログラムを変更できます。

Listing 1.238: assert() を追加

```
const char* get_month1_checked (int month)
```

```
{
    assert (month<12);
    return month1[month];
};
```

アサーションマクロは関数の開始時に妥当な値かチェックし、式が偽の場合に失敗します。

Listing 1.239: 最適化 MSVC 2013 x64

```
$SG3143 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '.', 00H
          DB      'c', 00H, 00H, 00H
$SG3144 DB      'm', 00H, 'o', 00H, 'n', 00H, 't', 00H, 'h', 00H, '<', 00H
          DB      '1', 00H, '2', 00H, 00H, 00H

month$ = 48
get_month1_checked PROC
$LN5:
    push     rbx
    sub     rsp, 32
    movsxd  rbx, ecx
    cmp     ebx, 12
    jl      SHORT $LN3@get_month1
    lea     rdx, OFFSET FLAT:$SG3143
    lea     rcx, OFFSET FLAT:$SG3144
    mov     r8d, 29
    call    _wassert
$LN3@get_month1:
    lea     rcx, OFFSET FLAT:month1
    mov     rax, QWORD PTR [rcx+rbx*8]
    add     rsp, 32
    pop     rbx
    ret     0
get_month1_checked ENDP
```

実際、assert() は関数ではなくマクロです。条件をチェックし、行数とファイル名を他の関数に渡してユーザに情報を報告します。

ファイル名と条件の両方がUTF-16でエンコードされています。行数も渡されます（29です）。

このメカニズムはおそらくすべてのコンパイラで同じです。GCCはこのようにします。

Listing 1.240: 最適化 GCC 4.9 x64

```
.LC1:
    .string "month.c"
.LC2:
    .string "month<12"

get_month1_checked:
    cmp     edi, 11
    jg      .L6
    movsx   rdi, edi
    mov     rax, QWORD PTR month1[0+rdi*8]
    ret
```

```
.L6:
    push    rax
    mov     ecx, OFFSET FLAT: __PRETTY_FUNCTION__ .2423
    mov     edx, 29
    mov     esi, OFFSET FLAT: .LC1
    mov     edi, OFFSET FLAT: .LC2
    call    __assert_fail

__PRETTY_FUNCTION__ .2423:
    .string "get_month1_checked"
```

GCCのマクロは利便性のために関数名も渡します。

何事もただではできませんが、サニタイズチェックもこれと同様です。

それはプログラムを遅くしますが、特にassert() マクロが小さなタイムクリティカルな関数で使用されると遅くなります。

なのでMSVCでは、例えばデバッグビルドではチェックを残し、リリースビルドでは取り除いたりします。

マイクロソフトWindows NTカーネルは「チェックされた」と「フリー」ビルドです。¹³¹

最初のは妥当性チェック（「チェックされた」なので）があり、もう一つはチェックしていません（チェックが「フリー」なので）。

もちろん、「チェックされた」カーネルはこれらのチェックのために遅く動作するので、通常はデバッグセッションでのみ使用されます。

特定の文字へのアクセス

文字列へのポインタの配列はこのようにアクセスできます。

```
#include <stdio.h>

const char* month[]=
{
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
};

int main()
{
    // 4番目の月、5番目のcharacter
    printf ("%c\n", month[3][4]);
};
```

...month[3] 式は *const char** 型をもつので、5番目の文字列はこのアドレスに4バイトを足した式から取得します。

さて、main() 関数に渡された引数リストは同じデータ型を持ちます。

¹³¹[msdn.microsoft.com/en-us/library/windows/hardware/ff543450\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff543450(v=vs.85).aspx)

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf ("第3引数, 2番目のcharacter: %c\n", argv[3][1]);
};
```

似た構文ですが、2次元配列とは異なることを理解することが非常に重要です。これについては後で検討します。

もう1つの重要なことに注意してください。アドレス指定される文字列は、各文字がASCII¹³²や拡張ASCIIのように1バイトを占めるシステムでエンコードされなければなりません。UTF-8はここでは動作しません。

第1.20.6節多次元配列

内部的には、多次元配列は本質的には一次元の配列と同じです。

コンピュータメモリは一次元なので、メモリは一次元配列です。便宜上、多次元配列は一次元として表現可能です。

例えば、3x4の配列の要素が12のセルの1次元配列にどのように配置されるかを示します。

Offset in memory	array element
0	[0][0]
1	[0][1]
2	[0][2]
3	[0][3]
4	[1][0]
5	[1][1]
6	[1][2]
7	[1][3]
8	[2][0]
9	[2][1]
10	[2][2]
11	[2][3]

表 1.3: 1次元配列としてメモリ上で表現される2次元配列

3*4配列の各セルがメモリ上でどう配置されるかを示します。

0	1	2	3
4	5	6	7
8	9	10	11

表 1.4: 2次元配列の各セルのメモリアドレス

¹³²American Standard Code for Information Interchange

したがって、必要な要素のアドレスを計算するには、まず最初のインデックスに 4（配列の幅）を掛けてから2番目のインデックスを追加します。これは行優先順位と呼ばれ、配列と行列表現のこの方法は、少なくとも C/C++ と Python で使用されます。単純な英単語の行優先順位は、「最初に、最初の行の要素を書き、次に2番目の行 ...最後に最後の行の要素を書き込む」という意味です。

表現のもう1つの方法は、列優先順位（配列の添字は逆順で使用されます）と呼ばれ、少なくとも Fortran、MATLAB、および R で使用されます。列優先順位は、単純な英語では、「最初に、最初の列の要素を書き込み、次に2番目の列を ...最後に最後の列の要素を書き込む」となります。

どの方法が良いでしょうか？

一般に、パフォーマンスとキャッシュメモリの観点からは、データ編成のための最良の方法は、要素が順次アクセスされる方法です。

したがって、関数が行ごとにデータにアクセスする場合は、行優先順位が優れていて、逆もまた同様です。

2次元配列の例

`char` 型の配列で作業していきます。これは、各要素がメモリ上に1バイトしか必要ないことを意味します。

行を埋める例

2行目を0～3の値で埋めてみましょう。

Listing 1.241: 行を埋める例

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // 配列のクリア
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // 2番目の行を0..3で満たす
    for (y=0; y<4; y++)
        a[1][y]=y;
};
```

3つの行はすべて赤でマークしてあります。2行目は0,1,2と3の値を持っています。

Address	Hex dump
00C33370	00 00 00 00 00 01 02 03 00 00 00 00 00 00 00 00
00C33380	02 00 00 00 C3 66 47 4E C3 66 47 4E 00 00 00 00
00C33390	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00C333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

図 1.92: OllyDbg: 配列が埋められる

列を埋める例

3列目を値0～2で埋めてみましょう。

Listing 1.242: 列を埋める例

```
#include <stdio.h>

char a[3][4];

int main()
{
    int x, y;

    // 配列のクリア
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;

    // 3番目のカラムを0..2で満たす
    for (x=0; x<3; x++)
        a[x][2]=x;
};
```

3つの行はここでも赤でマークしてあります。

各行の3番目の値が0,1と2で書かれています。

Address	Hex dump
01033380	00 00 00 00 00 00 01 00 00 00 02 00 02 00 00 00
01033390	00 00 00 00 1E 44 EF 31 1E 44 EF 31 00 00 00 00
010333A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
010333B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

図 1.93: OllyDbg: 配列が埋められる

2次元配列を1次元配列としてアクセスする

少なくとも2つの方法で、2次元配列を1次元配列としてアクセスすることが可能だといえます。

```
#include <stdio.h>

char a[3][4];

char get_by_coordinates1 (char array[3][4], int a, int b)
```

```

{
    return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
    // 入力された配列を1次元として扱う
    // 4は配列の幅
    return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
    // 入力された配列をポインタとして扱う
    // アドレスを計算し、値を得る
    // 4は配列の幅
    return *(array+a*4+b);
};

int main()
{
    a[2][3]=123;
    printf ("%d\n", get_by_coordinates1(a, 2, 3));
    printf ("%d\n", get_by_coordinates2(a, 2, 3));
    printf ("%d\n", get_by_coordinates3(a, 2, 3));
};

```

コンパイルして実行してください。¹³³ 正しい値を表示します。

MSVC 2013の結果は興味部会です。3つのルーチンはすべて同じです！

Listing 1.243: 最適化 MSVC 2013 x64

```

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
; RCX=配列のアドレス
; RDX=a
; R8=b
    movsxd    rax, r8d
; EAX=b
    movsxd    r9, edx
; R9=a
    add       rax, rcx
; RAX=b+配列のアドレス
    movzx     eax, BYTE PTR [rax+r9*4]
; AL= RAX+R9*4アドレスのバイトをロード = b+配列+a*4のアドレス = 配列+a*4+bのアドレス
    ret      0
get_by_coordinates3 ENDP
array$ = 8

```

¹³³ プログラムはC++ではなく、Cプログラムとしてコンパイルされます。.c拡張子でファイルを保存してMSVCでコンパイルします

```

a$ = 16
b$ = 24
get_by_coordinates2 PROC
    movsxd    rax, r8d
    movsxd    r9, edx
    add       rax, rcx
    movzx     eax, BYTE PTR [rax+r9*4]
    ret       0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
    movsxd    rax, r8d
    movsxd    r9, edx
    add       rax, rcx
    movzx     eax, BYTE PTR [rax+r9*4]
    ret       0
get_by_coordinates1 ENDP

```

GCCも同じルーチンを生成しますが、少し異なります。

Listing 1.244: 最適化 GCC 4.9 x64

```

; RDI=配列のアドレス
; RSI=a
; RDX=b

get_by_coordinates1:
; 32ビットint値の"a" と"b" を64ビットの符号付きintに拡張
    movsx    rsi, esi
    movsx    rdx, edx
    lea      rax, [rdi+rsi*4]
; RAX=RDI+RSI*4=配列のアドレス+a*4
    movzx    eax, BYTE PTR [rax+rdx]
; AL=RAX+RDXアドレスのバイトをロード=配列のアドレス+a*4+b
    ret

get_by_coordinates2:
    lea      eax, [rdx+rsi*4]
; RAX=RDX+RSI*4=b+a*4
    cdqe
    movzx    eax, BYTE PTR [rdi+rax]
; AL=RDI+RAXアドレスのバイトをロード=配列のアドレス+b+a*4
    ret

get_by_coordinates3:
    sal      esi, 2
; ESI=a<<2=a*4
; 32ビットint値の"a*4" と"b" を64ビットの符号付きintに拡張
    movsx    rdx, edx
    movsx    rsi, esi
    add      rdi, rsi

```

```

; RDI=RDI+RSI=配列のアドレス+a*4
    movzx    eax, BYTE PTR [rdi+rdx]
; AL=RDI+RDXアドレスのバイトをロード=配列のアドレス+a*4+b
    ret

```

3次元配列の例

多次元配列でも同じです。

int 型の配列で作業していきます。各要素はメモリ上で4バイト必要とします。

見てみましょう。

Listing 1.245: 単純な例

```

#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
    a[x][y][z]=value;
};

```

x86

MSVC 2010の結果

Listing 1.246: MSVC 2010

```

_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8           ; size = 4
_y$ = 12          ; size = 4
_z$ = 16          ; size = 4
_value$ = 20      ; size = 4
_insert   PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _x$[ebp]
    imul    eax, 2400           ; eax=600*4*x
    mov     ecx, DWORD PTR _y$[ebp]
    imul    ecx, 120           ; ecx=30*4*y
    lea     edx, DWORD PTR _a[eax+ecx] ; edx=a + 600*4*x + 30*4*y
    mov     eax, DWORD PTR _z$[ebp]
    mov     ecx, DWORD PTR _value$[ebp]
    mov     DWORD PTR [edx+eax*4], ecx ; *(edx+z*4)=値
    pop     ebp
    ret     0

```

```
_insert    ENDP
_TEXT      ENDS
```

特別なことはありません。インデックスの計算では、式 $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$ では3つの入力引数を使用され、配列を多次元として表現しています。*int* 型は32ビット（4バイト）なので、係数は4倍する必要があることを忘れないでください。

Listing 1.247: GCC 4.4.1

```
insert      public insert
            proc near

x           = dword ptr 8
y           = dword ptr 0Ch
z           = dword ptr 10h
value       = dword ptr 14h

            push    ebp
            mov     ebp, esp
            push    ebx
            mov     ebx, [ebp+x]
            mov     eax, [ebp+y]
            mov     ecx, [ebp+z]
            lea     edx, [eax+eax]      ; edx=y*2
            mov     eax, edx           ; eax=y*2
            shl     eax, 4              ; eax=(y*2)<<4 = y*2*16 = y*32
            sub     eax, edx           ; eax=y*32 - y*2=y*30
            imul    edx, ebx, 600      ; edx=x*600
            add     eax, edx           ; eax=eax+edx=y*30 + x*600
            lea     edx, [eax+ecx]     ; edx=y*30 + x*600 + z
            mov     eax, [ebp+value]
            mov     dword ptr ds:a[edx*4], eax ; *(a+edx*4)=value
            pop     ebx
            pop     ebp
            retn
insert      endp
```

GCCコンパイラは異なります。

計算での演算において $(30y)$ 、GCCは乗算命令を使わないコードを生成します。このようにします。 $(y+y) \ll 4 - (y+y) = (2y) \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. 従って、 $30y$ の計算には、加算命令が1つだけです。ビットシフト演算と減算が使用されます。これはより高速です。

ARM + 非最適化 Xcode 4.6.3 (LLVM) (Thumbモード)

Listing 1.248: 非最適化 Xcode 4.6.3 (LLVM) (Thumbモード)

```
_insert
value      = -0x10
z          = -0xC
```

```

y      = -8
x      = -4

; int型変数4つ分のローカルスタックの場所を確保
SUB     SP, SP, #0x10
MOV     R9, 0xFC2 ; a
ADD     R9, PC
LDR.W   R9, [R9] ; 配列へのポインタを取得
STR     R0, [SP, #0x10+x]
STR     R1, [SP, #0x10+y]
STR     R2, [SP, #0x10+z]
STR     R3, [SP, #0x10+value]
LDR     R0, [SP, #0x10+value]
LDR     R1, [SP, #0x10+z]
LDR     R2, [SP, #0x10+y]
LDR     R3, [SP, #0x10+x]
MOV     R12, 2400
MUL.W   R3, R3, R12
ADD     R3, R9
MOV     R9, 120
MUL.W   R2, R2, R9
ADD     R2, R3
LSLS    R1, R1, #2 ; R1=R1<<2
ADD     R1, R2
STR     R0, [R1] ; R1 - 配列要素のアドレス
; int型変数4つ分のローカルスタックのチャンクを開放
ADD     SP, SP, #0x10
BX      LR

```

非最適化 LLVMは変数すべてをローカルスタックに保存しますが、冗長です。

配列の要素のアドレスはすでに見た式によって計算されます。

ARM + 最適化 Xcode 4.6.3 (LLVM) (Thumbモード)

Listing 1.249: 最適化 Xcode 4.6.3 (LLVM) (Thumbモード)

```

_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1, R1, R1, LSL#4 ; R1 - y. R1=y<<4 - y = y*16 - y = y*15
ADD     R9, PC
LDR.W   R9, [R9] ; R9 = 配列へのポインタ
MLA.W   R0, R0, R12, R9 ; R0 - x, R12 - 2400, R9 - aへのポインタ。R0=x*2400 +
aへのポインタ
ADD.W   R0, R0, R1, LSL#3 ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + aへのポインタ +
y*15*8 =
; aへのポインタ + y*30*4 + x*600*4
STR.W   R3, [R0, R2, LSL#2] ; R2 - z, R3 - 値。address=R0+z*4 =
; aへのポインタ + y*30*4 + x*600*4 + z*4
BX      LR

```

既に見たシフト、加減算による乗算を置き換えるためのトリックもここにあります。

新しい命令を見てみます：RSB (*Reverse Subtract*)

単純に SUB として機能しますが、実行前にオペランドをスワップします。なぜでしょう？ SUB および RSB は、シフト係数が適用される第2のオペランド (LSL#4) への命令です。

ただし、この係数は第2オペランドにのみ適用されます。

これは、加算や乗算のような可換的な (交換可能な) 演算の場合は問題ありません。(結果を変更せずにオペランドを入れ替えてもかまいません)

しかし、減算は非可換的な演算なので、RSB が存在します。

MIPS

私の例はとても小さいので、GCCコンパイラはグローバルポインタによってアドレス可能な64KiB領域に配列を配置することに決めました。

Listing 1.250: 最適化 GCC 4.4.5 (IDA)

```
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=value
        sll    $v0, $a0, 5
; $v0 = $a0<<5 = x*32
        sll    $a0, 3
; $a0 = $a0<<3 = x*8
        addu   $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
        sll    $v1, $a1, 5
; $v1 = $a1<<5 = y*32
        sll    $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
        sll    $a1, 1
; $a1 = $a1<<1 = y*2
        subu   $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
        subu   $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
        la     $gp, __gnu_local_gp
        addu   $a0, $a1, $a0
; $a0 = $a1+$a0 = y*30+x*600
        addu   $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; テーブルのアドレスをロード
        lw     $v0, (a & 0xFFFF)($gp)
; インデックスを4倍して配列要素を検索
        sll    $a0, 2
; 掛け算したインデックスとテーブルアドレスを足し合わせる
        addu   $a0, $v0, $a0
; 値をテーブルに保存しリターン
```

```

jr      $ra
sw      $a3, 0($a0)

.comm a:0x1770

```

More examples

コンピュータ画面は2D配列として表現されますが、ビデオバッファは1次元配列です。これについてはこちらで：[?? on page ??](#)

本書での他の例としてはマインスイーパーゲームがあります。そのフィールドは2次元配列です：[?? on page ??](#)

第1.20.7節2次元配列としての文字列のパック

月の名前を返す関数を再考してみましょう：[リスト1.229](#)

月の名前の文字列へのポインタを準備するには少なくともメモリロード演算が1つ必要です。

メモリロード演算を取り除くことは可能でしょうか？

実際できます。文字列のリストを2次元配列として表現すれば。

```

#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
    { 'J','a','n','u','a','r','y', 0, 0, 0 },
    { 'F','e','b','r','u','a','r','y', 0, 0 },
    { 'M','a','r','c','h', 0, 0, 0, 0, 0 },
    { 'A','p','r','i','l', 0, 0, 0, 0, 0 },
    { 'M','a','y', 0, 0, 0, 0, 0, 0 },
    { 'J','u','n','e', 0, 0, 0, 0, 0, 0 },
    { 'J','u','l','y', 0, 0, 0, 0, 0, 0 },
    { 'A','u','g','u','s','t', 0, 0, 0, 0 },
    { 'S','e','p','t','e','m','b','e','r', 0 },
    { 'O','c','t','o','b','e','r', 0, 0, 0 },
    { 'N','o','v','e','m','b','e','r', 0, 0 },
    { 'D','e','c','e','m','b','e','r', 0, 0 }
};

// 0~11の範囲で
const char* get_month2 (int month)
{
    return &month2[month][0];
};

```

このような結果を得ました。

Listing 1.251: 最適化 MSVC 2013 x64


```

month2 DB      04aH
        DB      061H
        DB      06eH
        DB      075H
        DB      061H
        DB      072H
        DB      079H
        DB      00H
        DB      00H
        DB      00H
...

get_month2 PROC
; 符号拡張された入力引数を64ビット値に
    movsxd    rax, ecx
    lea       rcx, QWORD PTR [rax+rax*4]
; RCX=month+month*4=month*5
    lea       rax, OFFSET FLAT:month2
; RAX=テーブルへのポインタ
    lea       rax, QWORD PTR [rax+rcx*2]
; RAX=テーブルへのポインタ + RCX*2=テーブルへのポインタ + month*5*2=テーブルへのポ
    インタ + month*10
    ret       0
get_month2 ENDP

```

メモリアクセスは全くありません。

この関数でやっていることは、月の名前の最初の文字のポインタを計算することです：
 $pointer_to_the_table + month * 10$.

LEA 命令も2つあります。いくつかの MUL と MOV 命令として機能します。

配列の幅は10バイトです。

実際、ここでの最も長い文字列、「September」、は9バイトで、加えて0終端して10バイトです。

月の名前の残りはゼロで埋められて、月の名前は同じ領域（10バイト）を占有します。

従って、関数はより早く機能します。文字列の開始アドレスが簡単に計算できるためです。

最適化 GCC 4.9はより短くなります。

Listing 1.252: 最適化 GCC 4.9 x64

```

movsx    rdi, edi
lea      rax, [rdi+rdi*4]
lea      rax, month2[rax+rax]
ret

```

LEA は10倍するためにここでも使用されます。

最適化されていないコンパイラは、異なる方法で乗算を行います。

Listing 1.253: 非最適化 GCC 4.9 x64

```

get_month2:

```

```

        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        movsx   rdx, eax
; RDX = 符号拡張された入力値
        mov     rax, rdx
; RAX = month
        sal     rax, 2
; RAX = month<<2 = month*4
        add     rax, rdx
; RAX = RAX+RDX = month*4+month = month*5
        add     rax, rax
; RAX = RAX*2 = month*5*2 = month*10
        add     rax, OFFSET FLAT:month2
; RAX = month*10 + テーブルへのポインタ
        pop     rbp
        ret

```

非最適化 MSVCは単に IMUL 命令を使用します。

Listing 1.254: 非最適化 MSVC 2013 x64

```

month$ = 8
get_month2 PROC
        mov     DWORD PTR [rsp+8], ecx
        movsxd  rax, DWORD PTR month$[rsp]
; RAX = 符号拡張された入力値を64ビット値に
        imul    rax, rax, 10
; RAX = RAX*10
        lea     rcx, OFFSET FLAT:month2
; RCX = テーブルへのポインタ
        add     rcx, rax
; RCX = RCX+RAX = テーブルへのポインタ +month*10
        mov     rax, rcx
; RAX = テーブルへのポインタ +month*10
        mov     ecx, 1
; RCX = 1
        imul    rcx, rcx, 0
; RCX = 1*0 = 0
        add     rax, rcx
; RAX = テーブルへのポインタ +month*10 + 0 = テーブルへのポインタ +month*10
        ret     0
get_month2 ENDP

```

しかし、奇妙なことが1つあります。なぜ、0で乗算し、最終結果に0を加算するのでしょうか？

これはコンパイラのコードジェネレータの癖のように見えますが、コンパイラのテストでは検出されませんでした。(結局のところ、結果のコードは正しく動作します) このようなコードを意図的に検討することで、読者がそのようなコンパイラ成果物に困惑すべきでないときがあることを理解するでしょう。

32ビットARM

最適化 Keil Thumbモードでは、乗算命令 MULS を使用します。

Listing 1.255: 最適化 Keil 6/2013 (Thumbモード)

```
; R0 = month
    MOVS    r1,#0xa
; R1 = 10
    MULS    r0,r1,r0
; R0 = R1*R0 = 10*month
    LDR     r1,|L0.68|
; R1 = テーブルへのポインタ
    ADDS    r0,r0,r1
; R0 = R0+R1 = 10*month + テーブルへのポインタ
    BX      lr
```

ARMモードでの 最適化 Keil は加算とシフト命令を使用します。

Listing 1.256: 最適化 Keil 6/2013 (ARMモード)

```
; R0 = month
    LDR     r1,|L0.104|
; R1 = テーブルへのポインタ
    ADD     r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = month*5
    ADD     r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = テーブルへのポインタ + month*5*2 = テーブルへのポインタ +
    month*10
    BX      lr
```

ARM64

Listing 1.257: 最適化 GCC 4.9 ARM64

```
; W0 = month
    sxtw    x0, w0
; X0 = 符号拡張された入力値
    adrp    x1, .LANCHOR1
    add     x1, x1, :lo12:.LANCHOR1
; X1 = テーブルへのポインタ
    add     x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
    add     x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = テーブルへのポインタ + X0*10
    ret
```

SXTW は32ビット入力値を64ビットにし、X0に保存する、符号拡張のために使用されます。

ADRP/ADD の命令の組はテーブルのアドレスをロードするために使用されます。

ADD 命令には乗算に役立つ LSL サフィックスもあります。

Listing 1.258: 最適化 GCC 4.4.5 (IDA)

```

                                .globl get_month2
get_month2:
; $a0=month
                                sll    $v0, $a0, 3
; $v0 = $a0<<3 = month*8
                                sll    $a0, 1
; $a0 = $a0<<1 = month*2
                                addu   $a0, $v0
; $a0 = month*2+month*8 = month*10
; テーブルへのアドレスをロード
                                la     $v0, month2
; テーブルアドレスと計算したインデックスを足し合わせてリターン
                                jr      $ra
                                addu   $v0, $a0

month2:
                                .ascii "January"<0>
                                .byte 0, 0
aFebruary:
                                .ascii "February"<0>
                                .byte 0
aMarch:
                                .ascii "March"<0>
                                .byte 0, 0, 0, 0
aApril:
                                .ascii "April"<0>
                                .byte 0, 0, 0, 0
aMay:
                                .ascii "May"<0>
                                .byte 0, 0, 0, 0, 0, 0
aJune:
                                .ascii "June"<0>
                                .byte 0, 0, 0, 0, 0
aJuly:
                                .ascii "July"<0>
                                .byte 0, 0, 0, 0, 0
aAugust:
                                .ascii "August"<0>
                                .byte 0, 0, 0
aSeptember:
                                .ascii "September"<0>
aOctober:
                                .ascii "October"<0>
                                .byte 0, 0
aNovember:
                                .ascii "November"<0>
                                .byte 0
aDecember:
                                .ascii "December"<0>
                                .byte 0, 0, 0, 0, 0, 0, 0, 0, 0

```

結論

これはテキスト文字列を保存するための昔ながらの技術です。あなたは、たとえば、Oracle RDBMS でそれを見つけることができます。現代のコンピュータで実行する価値があるかどうかは言い難いですが、配列の良い例であるため、この本に追加されました。

第1.20.8節結論

配列は、隣り合って配置されたメモリ内の値の束です。

構造体を含むあらゆる要素種別に当てはまります。

特定の配列要素へのアクセスは、そのアドレスの計算に過ぎません。

したがって、最初の要素の配列とアドレスへのポインタは同じことです。このため、`ptr[0]` と `*ptr` の式は C/C++ で同等です。Hex-Raysはしばしば最初のを2番目のものに置き換えることは興味深いことです。これは、配列全体へのポインタで動作するかどうかかわからないときに行い、これが単一変数へのポインタであると考えます。

第1.20.9節練習問題

- <http://challenges.re/62>
- <http://challenges.re/63>
- <http://challenges.re/64>
- <http://challenges.re/65>
- <http://challenges.re/66>

第1.21節特定のビットを操作する

多くの関数では、入力引数をビットフィールドのフラグとして定義します。

もちろん、それらを *bool* 型の変数の組に置換することは可能ですが、効率的ではありません。

第1.21.1節特定のビットチェック

x86

Win32 APIの例です。

```
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ,
↳ FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

次の結果を得ます。(MSVC 2010)

Listing 1.259: MSVC 2010

```
push    0
push    128                ; 00000080H
push    4
push    0
push    1
push    -1073741824        ; c0000000H
push    OFFSET $SG78813
call    DWORD PTR __imp__CreateFileA@28
mov     DWORD PTR _fh$[ebp], eax
```

WinNT.hを見てみましょう。

Listing 1.260: WinNT.h

```
#define GENERIC_READ          (0x80000000L)
#define GENERIC_WRITE         (0x40000000L)
#define GENERIC_EXECUTE       (0x20000000L)
#define GENERIC_ALL            (0x10000000L)
```

すべてがクリアです。GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000で、この値が CreateFile()¹³⁴関数への2番目の引数として使用されています。

CreateFile() はこれらのフラグをどのようにチェックしているのでしょうか？

Windows XP SP3 x86のKERNEL32.DLLを見てみると、私たちはこのコードの断片を CreateFileW で見つけます。

Listing 1.261: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429    test    byte ptr [ebp+dwDesiredAccess+3], 40h
.text:7C83D42D    mov     [ebp+var_8], 1
.text:7C83D434    jz      short loc_7C83D417
.text:7C83D436    jmp     loc_7C810817
```

ここでは、TEST 命令を参照していますが、第2引数全体を取るのではなく、最上位バイト (ebp+dwDesiredAccess+3) のみを取り出し、フラグ 0x40 (ここでは GENERIC_WRITE フラグを意味します) をチェックします。

TEST は基本的に AND と同じ命令ですが、結果を保存することはありません (CMP は SUB と同じですが、結果を保存しないことを思い出してください (1.9.4 on page 109))。

このコードフラグメントのロジックは次のとおりです。

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

AND 命令がこのビットを離れると、ZF フラグはクリアされ、JZ 条件ジャンプは実行されません。条件ジャンプは、dwDesiredAccess 変数に 0x40000000 ビットが存在しない場合にのみ実行されます。AND の結果は0であり、ZF が設定され、条件付きジャンプが実行されます。

GCC 4.4.1とLinuxで試してみましょう。

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
    int handle;

    handle=open ("file", O_RDWR | O_CREAT);
};
```

¹³⁴[msdn.microsoft.com/en-us/library/aa363858\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx)

次の結果を得ます。

Listing 1.262: GCC 4.4.1

```

main                public main
                   proc near
var_20              = dword ptr -20h
var_1C              = dword ptr -1Ch
var_4               = dword ptr -4

                   push    ebp
                   mov     ebp, esp
                   and     esp, 0FFFFFFF0h
                   sub     esp, 20h
                   mov     [esp+20h+var_1C], 42h
                   mov     [esp+20h+var_20], offset aFile ; "file"
                   call    _open
                   mov     [esp+20h+var_4], eax
                   leave
                   retn
main               endp

```

libc.so.6 ライブラリの `open()` 関数を見てみると、それは単なるシステムコールです。

Listing 1.263: `open()` (libc.so.6)

```

.text:000BE69B      mov     edx, [esp+4+mode] ; mode
.text:000BE69F      mov     ecx, [esp+4+flags] ; flags
.text:000BE6A3      mov     ebx, [esp+4+filename] ; filename
.text:000BE6A7      mov     eax, 5
.text:000BE6AC      int     80h                ; LINUX - sys_open

```

したがって、`open()` のビットフィールドは、Linuxカーネルのどこかでチェックされるようです。

もちろん、GlibcとLinuxカーネルのソースコードの両方をダウンロードするのは簡単ですが、それを使わないで問題を理解することに興味があります。

したがって、Linux 2.6では、`sys_open` システムコールが呼び出されると、制御は最終的に `do_sys_open` に渡され、そこから `do_filp_open()` 関数 (`fs/namei.c` のカーネルソースツリーにあります) に渡されます。

注意：引数をスタック経由で渡すのとは別に、レジスタのいくつかをレジスタに渡す方法もあります。これは `fastcall` (?? on page ??) とも呼ばれます。これは、引数の値を読み取るためにCPUがメモリ内のスタックにアクセスする必要がないため、より高速に動作します。GCCには `regparm`¹³⁵ というオプションがあります。これにより、レジスタ経由で渡すことができる引数の数を設定することができます。

Linux 2.6カーネルは `-mregparm=3` オプション¹³⁶ でコンパイルされます。¹³⁷

¹³⁵ ohse.de/uwe/articles/gcc-attributes.html#func-regparm

¹³⁶ kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f

¹³⁷ カーネルツリーの `arch/x86/include/asm/calling.h` ファイルも参照してください

これが意味するところは、最初の3つの引数はレジスタ EAX、EDX、ECX を経由し、残りはスタック経由で渡されるということです。もちろん、引数の数が3よりも少ない場合、設定されたレジスタの一部のみが使用されます。

ですから、Linux Kernel 2.6.31をダウンロードし、Ubuntuでコンパイルしてみてください。make vmlinux し、IDA で開き、do_filp_open() 関数を見つけましょう。以下を見てください（コメントは私のものです）。

Listing 1.264: do_filp_open() (linux kernel 2.6.31)

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5番目の引数)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1番目の引数)
                mov     [ebp+var_7C], edx ; pathname (2番目の引数)
                mov     [ebp+var_78], ecx ; open_flag (3番目の引数)
                jnz     short loc_C01EF684
                mov     ebx, ecx          ; ebx <- open_flag
```

GCCは最初の3つの引数の値をローカルスタックに保存します。これが行われなかった場合、コンパイラはこれらのレジスタに触れず、コンパイラのregister allocatorには厳しい環境になります。

このコードの断片を見てみましょう：

Listing 1.265: do_filp_open() (linux kernel 2.6.31)

```
loc_C01EF684:    ; CODE XREF: do_filp_open+4F
                test    bl, 40h          ; 0_CREAT
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
                test    ebx, 10000h
                jz      short loc_C01EF6D3
                or      edi, 2
```

0x40 は 0_CREAT マクロと同じことです。open_flag は 0x40 としてチェックされ、このビットが1の場合、次の JNZ 命令が実行されます。

ARM

0_CREAT ビットはLinuxカーネル3.8.0ではチェックは異なります。

Listing 1.266: linux kernel 3.8.0

```

struct file *do_filp_open(int dfd, struct filename *pathname,
                          const struct open_flags *op)
{
    ...
    filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
    ...
}

static struct file *path_openat(int dfd, struct filename *pathname,
                                struct nameidata *nd, const struct open_flags *op, int ↵
                                ↵ flags)
{
    ...
    error = do_last(nd, &path, file, op, &opened, pathname);
    ...
}

static int do_last(struct nameidata *nd, struct path *path,
                  struct file *file, const struct open_flags *op,
                  int *opened, struct filename *name)
{
    ...
    if (!(open_flag & O_CREAT)) {
        ...
        error = lookup_fast(nd, path, &inode);
        ...
    } else {
        ...
        error = complete_walk(nd);
    }
    ...
}

```

ARMモード用にコンパイルされたカーネルが [IDA](#) でどのように見えるかは次のとおりです。

Listing 1.267: do_last() from vmlinux (IDA)

```

...
.text:C0169EA8    MOV        R9, R3    ; R3 - (4th argument) open_flag
...
.text:C0169ED4    LDR        R6, [R9]  ; R6 - open_flag
...
.text:C0169F68    TST        R6, #0x40 ; jumptable C0169F00 default case
.text:C0169F6C    BNE        loc_C016A128
.text:C0169F70    LDR        R2, [R4, #0x10]
.text:C0169F74    ADD        R12, R4, #8
.text:C0169F78    LDR        R3, [R4, #0xC]
.text:C0169F7C    MOV        R0, R4
.text:C0169F80    STR        R12, [R11, #var_50]
.text:C0169F84    LDRB       R3, [R2, R3]

```

```

.text:C0169F88      MOV        R2, R8
.text:C0169F8C      CMP        R3, #0
.text:C0169F90      ORRNE     R1, R1, #3
.text:C0169F94      STRNE     R1, [R4,#0x24]
.text:C0169F98      ANDS     R3, R6, #0x200000
.text:C0169F9C      MOV        R1, R12
.text:C0169FA0      LDRNE     R3, [R4,#0x24]
.text:C0169FA4      ANDNE     R3, R3, #1
.text:C0169FA8      EORNE     R3, R3, #1
.text:C0169FAC      STR        R3, [R11,#var_54]
.text:C0169FB0      SUB        R3, R11, #-var_38
.text:C0169FB4      BL        lookup_fast
...
.text:C016A128      loc_C016A128    ; CODE XREF: do_last.isra.14+DC
.text:C016A128      MOV        R0, R4
.text:C016A12C      BL        complete_walk
...

```

TST は、x86の TEST 命令に似ています。lookup_fast() はあるケースでは実行され、もう一つのケースでは complete_walk() が実行されるという事実によって、このコードフラグメントを視覚的に「発見」することができます。do_last() 関数のソースコードに相当します。0_CREAT マクロはここでも 0x40 と同じです。

第1.21.2節特定ビットの設定とクリア

例:

```

#include <stdio.h>

#define IS_SET(flag, bit)      ((flag) & (bit))
#define SET_BIT(var, bit)     ((var) |= (bit))
#define REMOVE_BIT(var, bit)  ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};

int main()
{
    f(0x12340678);
};

```

x86**非最適化 MSVC**

次の結果を得ます。(MSVC 2010)

Listing 1.268: MSVC 2010

```
_rt$ = -4          ; size = 4
_a$ = 8           ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR _rt$[ebp], eax
    mov     ecx, DWORD PTR _rt$[ebp]
    or      ecx, 16384          ; 00004000H
    mov     DWORD PTR _rt$[ebp], ecx
    mov     edx, DWORD PTR _rt$[ebp]
    and     edx, -513          ; ffffffffH
    mov     DWORD PTR _rt$[ebp], edx
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f ENDP
```

OR 命令は、他の1ビットを無視して1ビットをレジスタに設定します。

AND は1ビットをリセットします。AND は1を除くすべてのビットをコピーするだけであると言えます。実際、2番目の AND オペランドでは、保存する必要があるビットのみが設定され、コピーしたくないビットは設定されません（ビットマスクでは0）。これは、ロジックを覚えるのが簡単な方法です。

まず、使用する定数のバイナリ形式を見てみましょう。

Inverted 0x200 is 0xFFFFDFF (0b11111111111111111111111101111111).

入力値は 0x12340678 (0b10010001101000000011001111000)。どのようにロードされるか見ていきます。

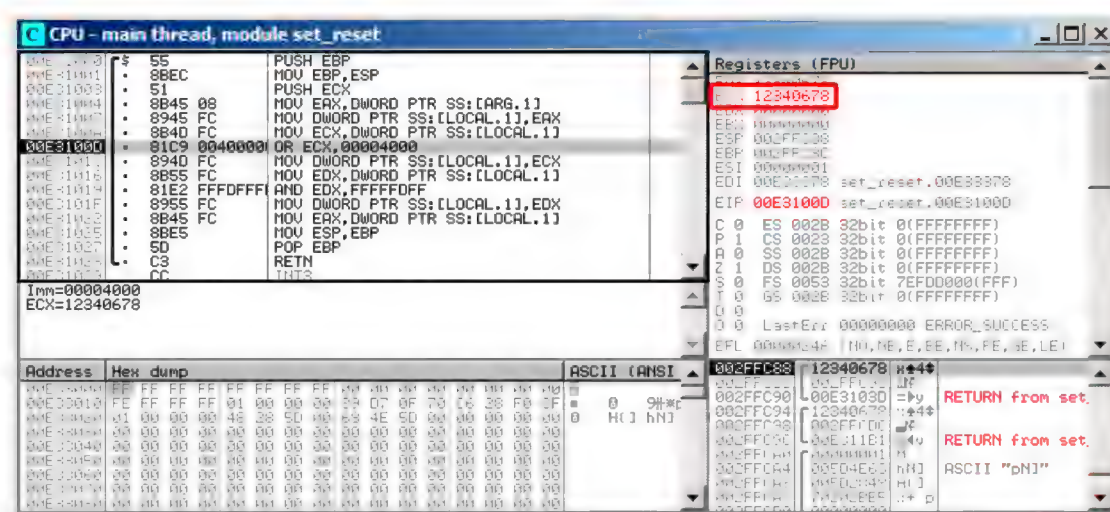


図 1.94: OllyDbg: 値が ECX にロード

OR が実行される。

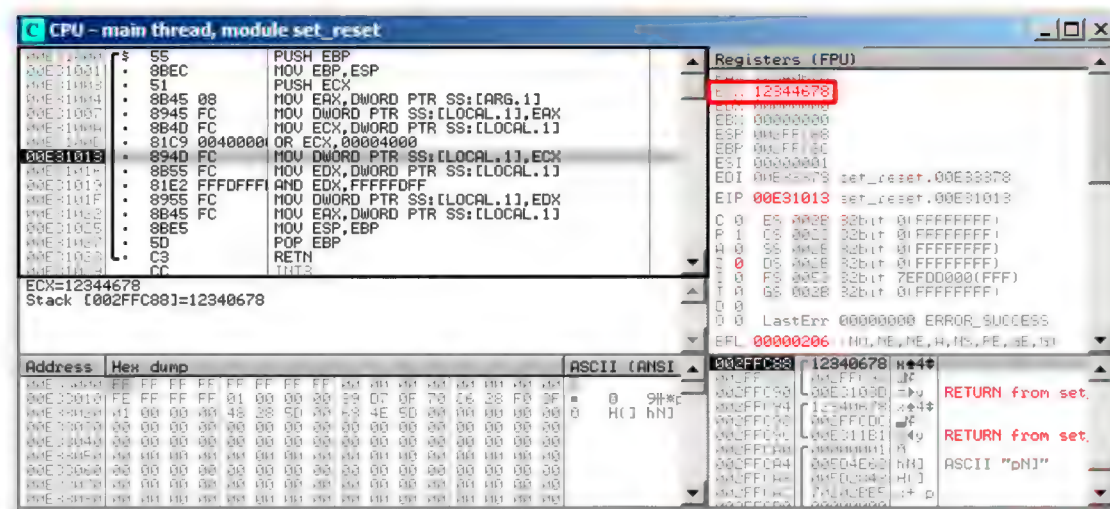


図 1.95: OllyDbg: OR が実行

15番目のビットがセット。0x1234**4**678 (0b10010001101000**1**00011001111000).

値がリロードされる（コンパイラが最適化モードではないから）。

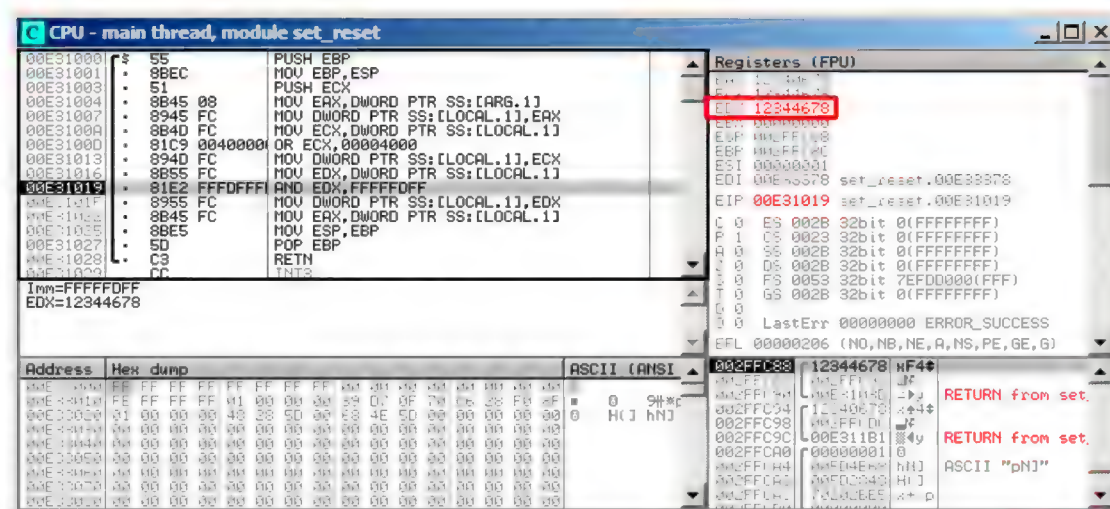


図 1.96: OllyDbg: 値が EDX にリロード

AND が実行される。

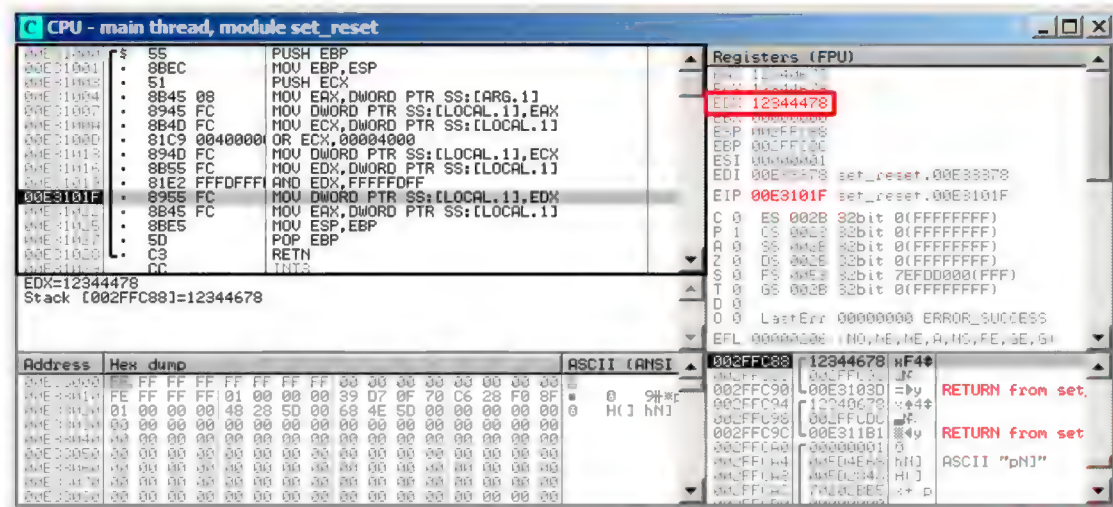


図 1.97: OllyDbg: AND が実行

10番目のビットがクリアされる。(または、言い換えると、10番目を除いてすべてのビットが残りました)そして、最終的な値は 0x12344478 (0b10010001101000100010001111000)です。

最適化 MSVC

MSVCで最適化を有効に (/Ox) してコンパイルすると、コードはもっと短くなります。

Listing 1.269: 最適化 MSVC

```

_a$ = 8 ; size = 4
_f PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513 ; ffffffffH
    or      eax, 16384 ; 00004000H
    ret     0
_f ENDP

```

非最適化 GCC

最適化なしのGCC 4.4.1を試してみましょう。

Listing 1.270: 非最適化 GCC

```

public f
proc near

```

```

var_4      = dword ptr -4
arg_0      = dword ptr 8

        push    ebp
        mov     ebp, esp
        sub     esp, 10h
        mov     eax, [ebp+arg_0]
        mov     [ebp+var_4], eax
        or      [ebp+var_4], 4000h
        and     [ebp+var_4], 0FFFFFFFh
        mov     eax, [ebp+var_4]
        leave
        retn
f        endp

```

冗長なコードが見られますが、非最適化MSVC版より短くなります。
最適化 -O3 を有効にしてGCCを試してみましょう。

最適化 GCC

Listing 1.271: 最適化 GCC

```

f        public f
        proc near
arg_0    = dword ptr 8

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_0]
        pop     ebp
        or      ah, 40h
        and     ah, 0FDh
        retn
f        endp

```

短くなります。より短いです。コンパイラがAHレジスタを介してEAXレジスタの部分で動作することは注目に値します。これは、8番目のビットから15番目のビットまでのEAXレジスタの部分です。

バイトの並び順							
第7	第6	第5	第4	第3	第2	第1	第0
RAX ^{x64}							
				EAX			
						AX	
						AH	AL

注意：16ビットCPU 8086アキュムレータはAXと命名され、8ビットの2つのレジスタで構成されていました。AL（下位バイト）およびAH（上位バイト）です。80386ではほとんどすべてのレジスタが32ビットに拡張されて、アキュムレータの名前はEAXでしたが、互換性のために古い部分にはAX/AH/ALとしてアクセスすることができます。

すべてのx86 CPUは16ビットの8086 CPUの後継バージョンなので、古い 16ビットのオペコードは新しい32ビットのものよりも短くなります。だから、`or ah, 40h` 命令は3バイトしか占有しません。ここでは `or eax, 04000h` を発行する方が論理的ですが、それは5または6バイトです。(最初のオペランドのレジスタが EAX でない場合)

最適化 GCC and regparm

-O3 最適化フラグをオンにして `regparm=3` に設定するとさらに短くなります。

Listing 1.272: 最適化 GCC

```
f      public f
      proc near
      push    ebp
      or      ah, 40h
      mov     ebp, esp
      and     ah, 0FDh
      pop     ebp
      retn
f      endp
```

実際、最初の引数はすでに EAX にロードされているので、インプレースで処理することは可能です。関数プロローグ (`push ebp / mov ebp, esp`) とエピローグ (`pop ebp`) はここでは簡単に省略することができますが、GCCはおそらくこのようなコードサイズの最適化を行うには不十分であることに注意してください。しかし、このような短い関数はインライン関数より優れています。(?? on page ??)

ARM + 最適化 Keil 6/2013 (ARMモード)

Listing 1.273: 最適化 Keil 6/2013 (ARMモード)

```
02 0C C0 E3      BIC      R0, R0, #0x200
01 09 80 E3      ORR      R0, R0, #0x4000
1E FF 2F E1      BX       LR
```

BIC (*Bitwise bit Clear*) は特定のビットをクリアする命令です。AND 命令に似ていますが、反転したオペランドを使用します。つまり、NOT +AND 命令ペアに類似しています。

ORR is 「logical or」, analogous to OR in x86.

ORR は「論理OR」です。x86の OR に類似しています。

ここまでは簡単です。

ARM + 最適化 Keil 6/2013 (Thumbモード)

Listing 1.274: 最適化 Keil 6/2013 (Thumbモード)

```
01 21 89 03      MOVS     R1, 0x4000
08 43            ORRS     R0, R1
49 11            ASRS     R1, R1, #5    ; 0x200を生成しR1に配置する
88 43            BICS     R0, R1
70 47            BX       LR
```

KeilはThumbモードのコードが 0x4000 から 0x200 になり、0x200 を任意のレジスタに書き込むコードよりもコンパクトであると判断したようです。

したがって、ASRS ([Japanese text placeholder](#)) の助けを借りて、この値は $0x4000 \gg 5$ として計算されます。

ARM + 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

Listing 1.275: 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

42 0C C0 E3	BIC	R0, R0, #0x4200
01 09 80 E3	ORR	R0, R0, #0x4000
1E FF 2F E1	BX	LR

LLVMが生成したコードは、次のようになるかもしれません。

```
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

そして、これはまさに必要としているものです。しかしなぜ 0x4200 なのでしょう。おそらく、LLVMの最適化マイザが生成した生成物でしょう。

138

コンパイラの最適化マイザのエラーかもしれませんが、生成されたコードはともあれ正しく動作します。

コンパイラのアノマリについての詳細はこちら ([?? on page ??](#))

Thumbモードでの 最適化 Xcode 4.6.3 (LLVM) は同じコードを生成します。

ARM: BIC 命令についての詳細

例を少し変更してみましょう。

```
int f(int a)
{
    int rt=a;

    REMOVE_BIT (rt, 0x1234);

    return rt;
};
```

ARMモードの最適化Keil 5.03 の結果は以下のようになります。

```
f PROC
    BIC    r0,r0,#0x1000
    BIC    r0,r0,#0x234
    BX     lr
ENDP
```

¹³⁸Apple Xcode 4.6.3にバンドルされたLLVM build 2410.2.00です

BIC 命令が2つあります。すなわち、ビット 0x1234 は2パスでクリアされます。

なぜなら1つの BIC 命令では 0x1234 をエンコードすることが不可能だからです。しかし、0x1000 と 0x234 をエンコードすることはできます。

ARM64: 最適化 GCC (Linaro) 4.9

最適化 GCCコンパイラでARM64をコンパイルするなら BIC の代わりに AND 命令を使用できます。

Listing 1.276: 最適化 GCC (Linaro) 4.9

```
f:
    and    w0, w0, -513      ; 0xFFFFFFFFFFFFFFDFF
    orr    w0, w0, 16384     ; 0x4000
    ret
```

ARM64: 非最適化 GCC (Linaro) 4.9

非最適化 GCC はもっと冗長なコードを生成しますが、最適化されたように動作します。

Listing 1.277: 非最適化 GCC (Linaro) 4.9

```
f:
    sub    sp, sp, #32
    str    w0, [sp,12]
    ldr    w0, [sp,12]
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    orr    w0, w0, 16384     ; 0x4000
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    and    w0, w0, -513      ; 0xFFFFFFFFFFFFFFDFF
    str    w0, [sp,28]
    ldr    w0, [sp,28]
    add    sp, sp, 32
    ret
```

MIPS

Listing 1.278: 最適化 GCC 4.4.5 (IDA)

```
f:
; $a0=a
                                ori    $a0, 0x4000
; $a0=a|0x4000
                                li     $v0, 0xFFFFDFF
                                jr      $ra
                                and     $v0, $a0, $v0
; 終了時: $v0 = $a0 & $v0 = a|0x4000 & 0xFFFFDFF
```

ORI はもちろん、OR演算を行います。命令の中の「I」は機械語の中に値が埋め込まれることを意味します。

しかしその後、私たちは AND があります。0xFFFFDFFを単一の命令に埋め込むことは不可能であるため、ANDI を使用する方法はありません。そのため、コンパイラは最初にレジスタ \$V0 に0xFFFFDFFをロードしてから、レジスタからすべての値を取る AND を生成します。

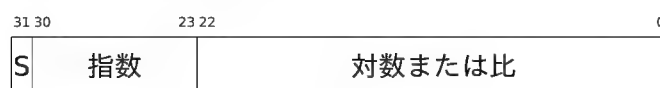
第1.21.3節シフト

C/C++ でのビットシフトは << と >> 演算子を使って実装されます。x86 ISA はシフトのためにSHL (SHift Left) と SHR (SHift Right) 命令を持っています。シフト命令はしばしば2のべき乗 2^n において除算や乗算が使用されます（例えば1,2,4,8など）。[1.18.1 on page 261](#)、[1.18.2 on page 266](#)。

シフト操作も非常に重要です。シフト演算は特定ビットの分離や複数の散在ビット値の構築に用いられることが多いからです。

第1.21.4節特定ビットのセットやクリア: **FPU** の例

IEEE 754形式で *float* 型がどのように配置されるのか見てみます。



(S — 記号)

数字の符号情報はMSB¹³⁹にあります。FPU命令なしで浮動小数点数の符号を変更することは可能でしょうか？

```
#include <stdio.h>

float my_abs (float i)
{
    unsigned int tmp=*(unsigned int*)&i & 0x7FFFFFFF;
    return *(float*)&tmp;
};

float set_sign (float i)
{
    unsigned int tmp=*(unsigned int*)&i | 0x80000000;
    return *(float*)&tmp;
};

float negate (float i)
{
    unsigned int tmp=*(unsigned int*)&i ^ 0x80000000;
    return *(float*)&tmp;
};
```

¹³⁹ 最上位ビット

```

int main()
{
    printf ("my_abs():\n");
    printf ("%f\n", my_abs (123.456));
    printf ("%f\n", my_abs (-456.123));
    printf ("set_sign():\n");
    printf ("%f\n", set_sign (123.456));
    printf ("%f\n", set_sign (-456.123));
    printf ("negate():\n");
    printf ("%f\n", negate (123.456));
    printf ("%f\n", negate (-456.123));
};

```

実際の変換をせずに *float* 値との間でコピーを行うには、C/C++ でこのトリックが必要です。したがって、3つの関数があります。my_abs() はMSBをリセットします。set_sign() はMSBを設定します。negate() はそれを反転させます。

XOR を使ってビットを反転することができます。

x86

コードはかなり簡単です。

Listing 1.279: 最適化 MSVC 2012

```

_tmp$ = 8
_i$ = 8
_my_abs PROC
    and     DWORD PTR _i$[esp-4], 2147483647 ; 7fffffffH
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_my_abs ENDP

_tmp$ = 8
_i$ = 8
_set_sign PROC
    or      DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_set_sign ENDP

_tmp$ = 8
_i$ = 8
_negate PROC
    xor     DWORD PTR _i$[esp-4], -2147483648 ; 80000000H
    fld     DWORD PTR _tmp$[esp-4]
    ret     0
_negate ENDP

```

float 型の入力値はスタックから取得されますが、整数値として扱われます。

AND と OR は望むビットをリセットそしてセットします。XOR は反転します。

最後に、変更された値が ST0 にロードされます。浮動小数点数はこのレジスタにリターンされるからです。

さて、x64向けの最適化MSVC 2012で試してみましょう。

Listing 1.280: 最適化 MSVC 2012 x64

```
tmp$ = 8
i$ = 8
my_abs PROC
    movss    DWORD PTR [rsp+8], xmm0
    mov     eax, DWORD PTR i$[rsp]
    btr     eax, 31
    mov     DWORD PTR tmp$[rsp], eax
    movss    xmm0, DWORD PTR tmp$[rsp]
    ret     0
my_abs ENDP
_TEXT    ENDS

tmp$ = 8
i$ = 8
set_sign PROC
    movss    DWORD PTR [rsp+8], xmm0
    mov     eax, DWORD PTR i$[rsp]
    bts     eax, 31
    mov     DWORD PTR tmp$[rsp], eax
    movss    xmm0, DWORD PTR tmp$[rsp]
    ret     0
set_sign ENDP

tmp$ = 8
i$ = 8
negate PROC
    movss    DWORD PTR [rsp+8], xmm0
    mov     eax, DWORD PTR i$[rsp]
    btc     eax, 31
    mov     DWORD PTR tmp$[rsp], eax
    movss    xmm0, DWORD PTR tmp$[rsp]
    ret     0
negate ENDP
```

入力値は XMM0 に渡され、そしてローカルスタックにコピーされて、新しい命令がいくつか見られます。BTR、BTS、BTC です。

各命令は特定のビットをリセット (BTR)、セット (BTS) そして反転 (または補数: BTC) するのに用いられます。0から数えて31番目のビットはMSBです。

最後に、結果は XMM0 にコピーされます。浮動小数点数の値はWin64の環境では XMM0 を通してリターンされるからです。

MIPS

GCC 4.4.5でMIPS向けのコードはほとんど同じです。

Listing 1.281: 最適化 GCC 4.4.5 (IDA)

```

my_abs:
; コプロセッサ1から移動
        mfc1    $v1, $f12
        li      $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; ANDを実行
        and     $v0, $v1
; コプロセッサ1に移動:
        mtc1    $v0, $f0
; リターン
        jr      $ra
        or      $at, $zero ; 分岐遅延スロット

set_sign:
; コプロセッサ1から移動
        mfc1    $v0, $f12
        lui     $v1, 0x8000
; $v1=0x80000000
; ORを実行
        or      $v0, $v1, $v0
; コプロセッサ1に移動:
        mtc1    $v0, $f0
; リターン
        jr      $ra
        or      $at, $zero ; 分岐遅延スロット

negate:
; コプロセッサ1から移動
        mfc1    $v0, $f12
        lui     $v1, 0x8000
; $v1=0x80000000
; XORを実行
        xor     $v0, $v1, $v0
; コプロセッサ1に移動
        mtc1    $v0, $f0
; リターン
        jr      $ra
        or      $at, $zero ; 分岐遅延スロット

```

単一の LUI 命令が使用され、レジスタに 0x80000000 がロードされます。LUI は低位 16 ビットをクリアし、ゼロにするので、後続に ORI がなくても LUI で十分です。

ARM

最適化 Keil 6/2013 (ARMモード)

Listing 1.282: 最適化 Keil 6/2013 (ARMモード)

```

my_abs PROC
; ビットをクリア
        BIC     r0, r0, #0x80000000

```

```

        BX      lr
    ENDP

set_sign PROC
; 0Rを実行
    ORR        r0,r0,#0x80000000
    BX         lr
    ENDP

negate PROC
; XORを実行
    EOR        r0,r0,#0x80000000
    BX         lr
    ENDP

```

ここまでは順調です。

ARMは BIC 命令があり、特定のビットを明示的にクリアします。EOR はARM命令で XOR のことです (「Exclusive OR」)。

最適化 Keil 6/2013 (Thumbモード)

Listing 1.283: 最適化 Keil 6/2013 (Thumbモード)

```

my_abs PROC
    LSLS        r0,r0,#1
; r0=i<<1
    LSRS        r0,r0,#1
; r0=(i<<1)>>1
    BX         lr
    ENDP

set_sign PROC
    MOVS        r1,#1
; r1=1
    LSLS        r1,r1,#31
; r1=1<<31=0x80000000
    ORRS        r0,r0,r1
; r0=r0 | 0x80000000
    BX         lr
    ENDP

negate PROC
    MOVS        r1,#1
; r1=1
    LSLS        r1,r1,#31
; r1=1<<31=0x80000000
    EORS        r0,r0,r1
; r0=r0 ^ 0x80000000
    BX         lr
    ENDP

```


ARMのThumbモードは16ビット命令を提供します。そんなに多くのデータをエンコードはできないので、MOVSL/LSLS 命令ペアが定数0x80000000を形づくるのに使用されます。このように動作します： $1 \ll 31 = 0x80000000$

my_abs のコードは奇妙で、この式のように効果的に機能します： $(i \ll 1) \gg 1$ この文は無意味に見えます。しかし、*input* $\ll 1$ が実行されると、MSB（符号ビット）がドロップされるだけです。後続の *result* $\gg 1$ 文が実行されると、すべてのビットが現在自分の場所にありますが、シフト演算から出現するすべての「新しい」ビットは常にゼロであるため、MSBはゼロになります。これが LSLS/LSRS 命令ペアがMSBをクリアする方法です。

最適化 GCC 4.6.3 (Raspberry Pi, ARMモード)

Listing 1.284: 最適化 GCC 4.6.3 for Raspberry Pi (ARMモード)

```
my_abs
; S0からR2にコピー
      FMRS    R2, S0
; ビットをクリア
      BIC     R3, R2, #0x80000000
; R3からS0にコピー
      FMSR    S0, R3
      BX     LR

set_sign
; S0からR2にコピー
      FMRS    R2, S0
; ORを実行
      ORR     R3, R2, #0x80000000
; R3からS0にコピー
      FMSR    S0, R3
      BX     LR

negate
; S0からR2にコピー
      FMRS    R2, S0
; ADDを実行
      ADD     R3, R2, #0x80000000
; R3からS0にコピー
      FMSR    S0, R3
      BX     LR
```

QEMUでRaspberry Pi Linuxを動かしてARM FPUをエミュレートしてみましょう。Rレジスタの代わりにSレジスタが浮動小数点数に使用されます。

FMRS 命令はGPRからFPUそして逆にもデータをコピーします。

my_abs() と set_sign() は期待通りに見えますが、negate() はどうでしょうか？XORの代わりに ADD があるのはどうしてでしょうか？

信じがたいかもしれませんが、命令 ADD register, 0x80000000 は XOR register, 0x80000000 のように動作します。まず、ゴールは何でしょうか？ゴールはMSBを反転させることなので、XOR 演算は忘れましょう。学校レベルの数学からは他の値に1000を加算する

ことは最後の3桁には影響しないと思うかもしれませんが。例えば： $1234567 + 10000 = 1244567$ (最後の4桁は影響を受けない)

しかし、ここではバイナリベースで操作すると、 $0x80000000$ は $0b10000000000000000000000000000000$ すなわち最高位のビットのみセットされます。

任意の値に $0x80000000$ を加算すると低位の31ビットに影響しませんが、MSBだけ影響します。0に1を加算すると結果は1です。

1に1を加算すると結果はバイナリ形式で $0b10$ になりますが、(0から数えて) 32番目のビットはドロップします。レジスタは32ビット幅なので、結果は0になります。XOR が ADD で置き換え可能なのはそのためです。

GCCがなぜこうすると決定したかはわかりませんが、正しく動作します。

第1.21.5節 Counting bits set to 1

入力値のビットの数を計算する関数の単純な例です。

この操作は「集団カウント」とも呼ばれます。¹⁴⁰

```
#include <stdio.h>

#define IS_SET(flag, bit)      ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};

int main()
{
    f(0x12345678); // test
};
```

このループでは、ループカウンタ値 i は0から31を数えます。 $1 \ll i$ 文は1から $0x80000000$ まで数えます。自然言語でこの操作を説明すると、1を n ビット左シフトするといえます。言い換ええると、 $1 \ll i$ 文は結果として32ビット数のすべての可能なビット位置を生成します。右側の解放されたビットは常にクリアされます。

$i = 0 \dots 31$ で取りうるすべての値の表です。

¹⁴⁰ (SSE4をサポートする) モダンx86 CPUはこのためにPOPCNT命令を持っています

C/C++ 表現	2のべき乗	10進数形式	16進数形式
1 << 0	2 ⁰	1	1
1 << 1	2 ¹	2	2
1 << 2	2 ²	4	4
1 << 3	2 ³	8	8
1 << 4	2 ⁴	16	0x10
1 << 5	2 ⁵	32	0x20
1 << 6	2 ⁶	64	0x40
1 << 7	2 ⁷	128	0x80
1 << 8	2 ⁸	256	0x100
1 << 9	2 ⁹	512	0x200
1 << 10	2 ¹⁰	1024	0x400
1 << 11	2 ¹¹	2048	0x800
1 << 12	2 ¹²	4096	0x1000
1 << 13	2 ¹³	8192	0x2000
1 << 14	2 ¹⁴	16384	0x4000
1 << 15	2 ¹⁵	32768	0x8000
1 << 16	2 ¹⁶	65536	0x10000
1 << 17	2 ¹⁷	131072	0x20000
1 << 18	2 ¹⁸	262144	0x40000
1 << 19	2 ¹⁹	524288	0x80000
1 << 20	2 ²⁰	1048576	0x100000
1 << 21	2 ²¹	2097152	0x200000
1 << 22	2 ²²	4194304	0x400000
1 << 23	2 ²³	8388608	0x800000
1 << 24	2 ²⁴	16777216	0x1000000
1 << 25	2 ²⁵	33554432	0x2000000
1 << 26	2 ²⁶	67108864	0x4000000
1 << 27	2 ²⁷	134217728	0x8000000
1 << 28	2 ²⁸	268435456	0x10000000
1 << 29	2 ²⁹	536870912	0x20000000
1 << 30	2 ³⁰	1073741824	0x40000000
1 << 31	2 ³¹	2147483648	0x80000000

このような定数（ビットマスク）はコード上、非常によく現れます。現役のリバースエンジニアはこれらを素早く見つけなければなりません。

65536以下の10進数と16進数は簡単に記憶できます。65536を超える10進数はおそらく記憶する価値はないでしょう。

これらの定数は、フラグを特定のビットにマッピングするために非常によく使用されます。たとえば、Apache 2.4.6のソースコードから `ssl_private.h` を抜粋した例を次に示します。

```
/**
 * Define the SSL options
 */
#define SSL_OPT_NONE          (0)
#define SSL_OPT_RELSET        (1<<0)
#define SSL_OPT_STDENVVARS    (1<<1)
```

```
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH (1<<4)
#define SSL_OPT_STRICTREQUIRE (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```

私たちの例に戻りましょう。

IS_SET マクロはビットの数を *a* でチェックします。

IS_SET マクロは実際、論理AND演算 (AND) で、特定のビットがそこになれば0を返すか、ビットが存在すれば、ビットをマスクします。C/C++ の *if()* 演算子は、その式がゼロでない場合に実行しますが、123456であっても正しく動作します。

x86

MSVC

MSVC 2010でコンパイルしてみましょう。

Listing 1.285: MSVC 2010

```
_rt$ = -8          ; size = 4
_i$ = -4           ; size = 4
_a$ = 8            ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8
    mov     DWORD PTR _rt$[ebp], 0
    mov     DWORD PTR _i$[ebp], 0
    jmp     SHORT $LN4@f
$LN3@f:
    mov     eax, DWORD PTR _i$[ebp]    ; iをインクリメント
    add     eax, 1
    mov     DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp     DWORD PTR _i$[ebp], 32     ; 00000020H
    jge     SHORT $LN2@f               ; ループ終了?
    mov     edx, 1
    mov     ecx, DWORD PTR _i$[ebp]
    shl     edx, cl                    ; EDX=EDX<<CL
    and     edx, DWORD PTR _a$[ebp]
    je      SHORT $LN1@f               ; AND命令の結果は0?
                                           ; そうなら次の命令をスキップ
    mov     eax, DWORD PTR _rt$[ebp]   ; そうでなければ、0ではない
    add     eax, 1                     ; rtをインクリメント
    mov     DWORD PTR _rt$[ebp], eax
$LN1@f:
    jmp     SHORT $LN3@f
$LN2@f:
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
```

```
    pop    ebp
    ret    0
_f      ENDP
```

OllyDbg

例を OllyDbg にロードしてみましょう。入力値は 0x12345678 です。

$i = 1$ の場合に、 i がどう ECX にロードされるかを見ていきます。

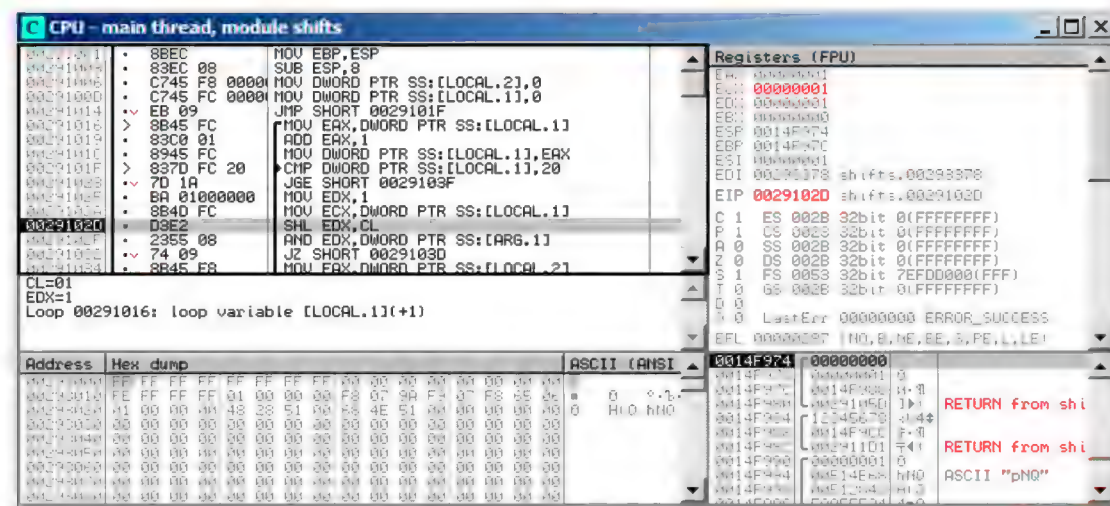


図 1.98: OllyDbg: $i = 1$ 、 i が ECX にロードされる

EDX は1です。SHL はたった今実行されます。

SHL が実行されました。

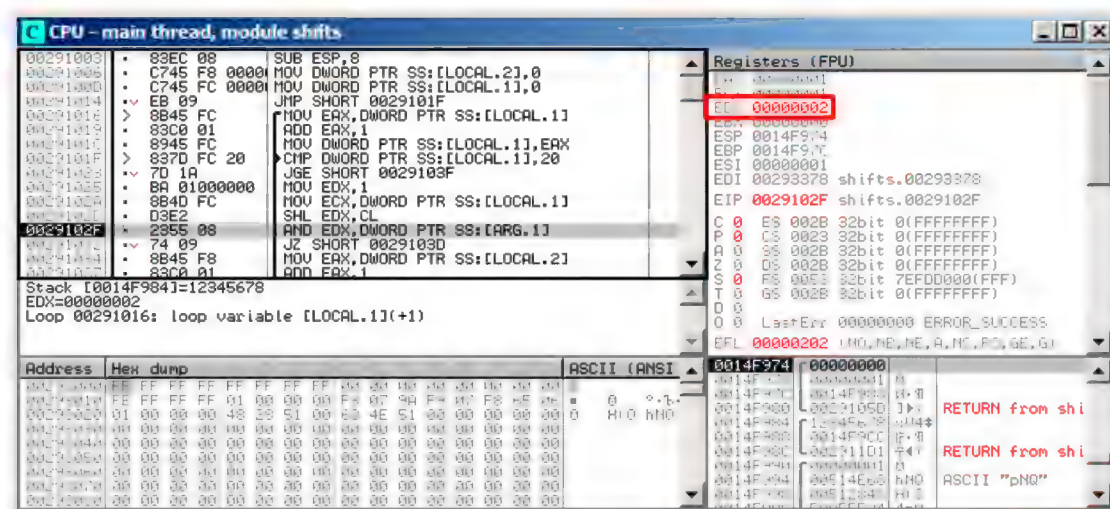


图 1.99: OllyDbg: $i = 1$ 、 $EDX = 1 \ll 1 = 2$

EDX は $1 \ll 1$ (または2) を含みます。これはビットマスクです。

AND は ZF を1にセットし、入力値 (0x12345678) を2でANDして結果が0になることを意味します。

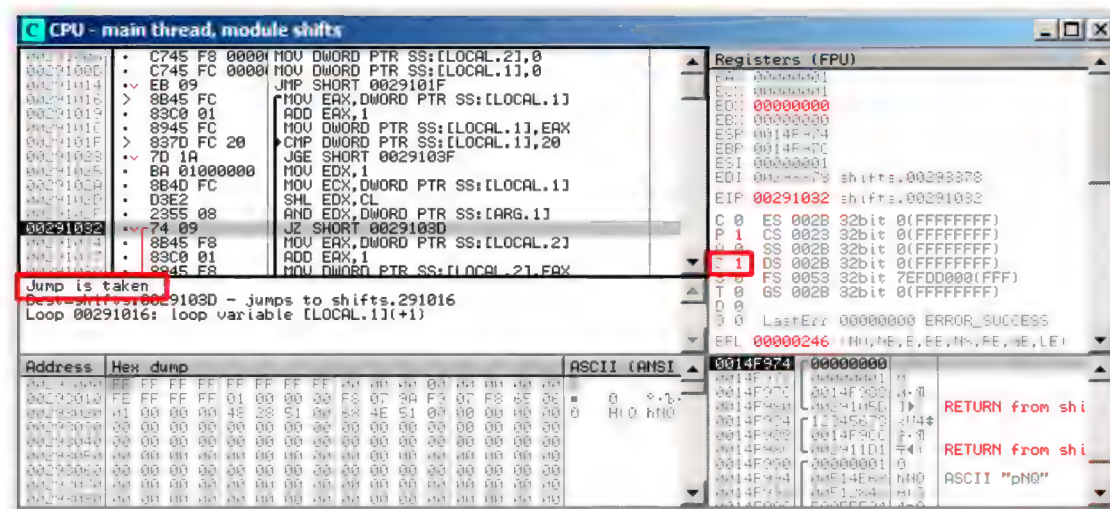


図 1.100: OllyDbg: $i = 1$ 、入力値にそのようなビットはありますか？いいえ (ZF = 1)

従って、入力値には対応するビットはありません。

カウンタをインクリメントするコード片は実行されません。JZ 命令はバイパスします。

もう少しトレースしてみましょう。 i は4です。SHL がここで実行されます。

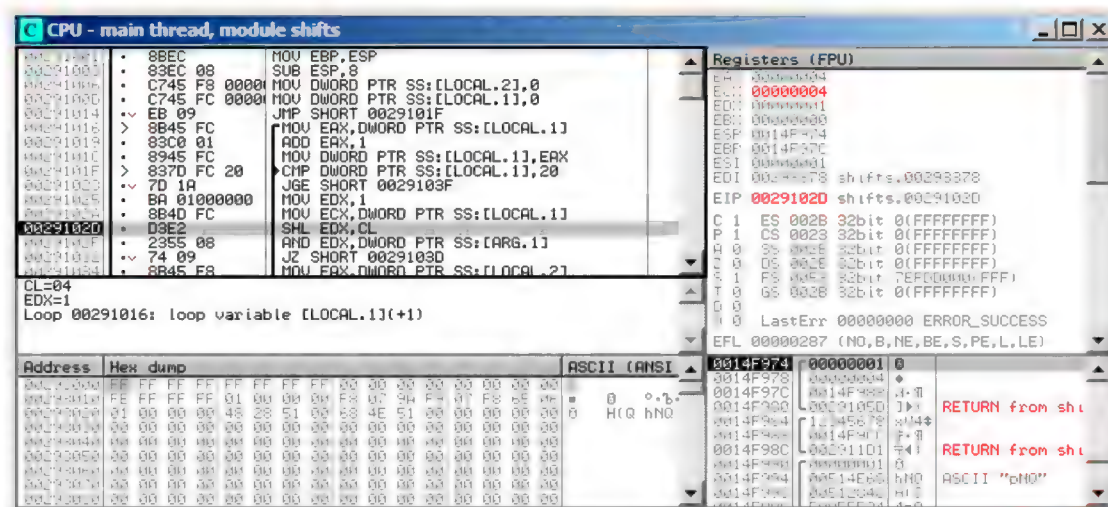


図 1.101: OllyDbg: $i = 4$ 、 i は ECX にロードされる

EDX = $1 \ll 4$ (または 0×10 もしくは 16):

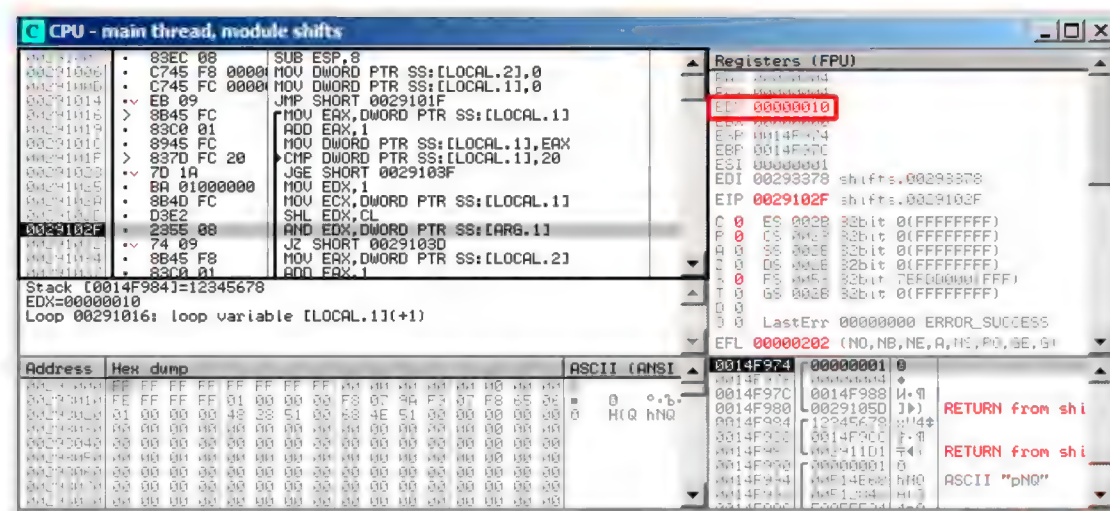


図 1.102: OllyDbg: $i = 4$ 、 $EDX = 1 \ll 4 = 0 \times 10$

これは別のビットマスクです。

AND が実行されます。

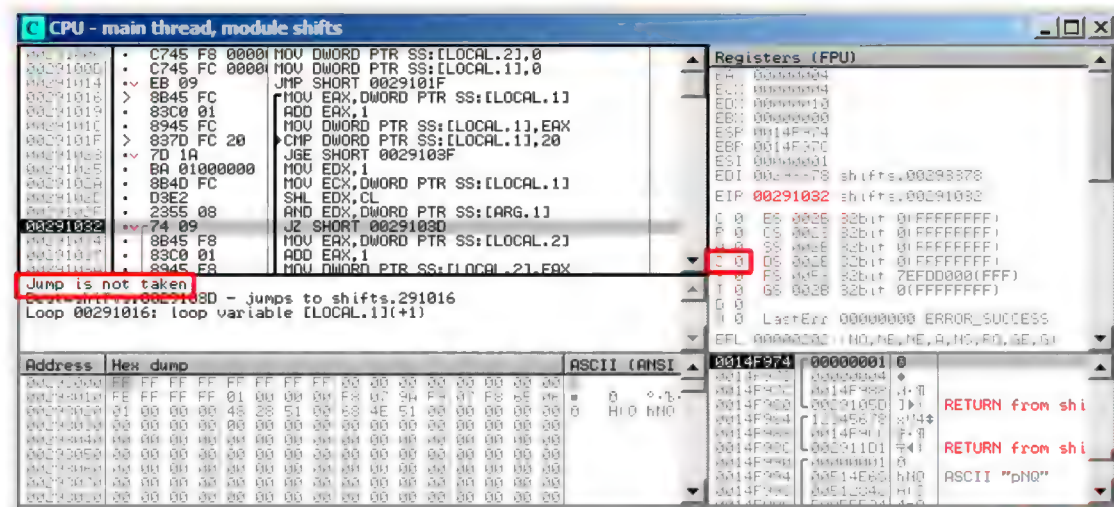


図 1.103: OllyDbg: $i = 4$ 、入力値にそのようなビットはありますか？はい (ZF = 0)

ZF は 0 です。このビットは入力値にあるからです。実際、 $0x12345678 \& 0x10 = 0x10$ です。

ジャンプは実行されず、ビットカウンタはインクリメントします

関数は 13 をリターンします。これは、 $0x12345678$ に設定されたビットの総数です。

GCC

GCC 4.4.1 でコンパイルしてみましょう。

Listing 1.286: GCC 4.4.1

```

f                public f
proc near
rt               = dword ptr -0Ch
i                = dword ptr -8
arg_0           = dword ptr 8

                push    ebp
                mov     ebp, esp
                push    ebx
                sub     esp, 10h
                mov     [ebp+rt], 0
                mov     [ebp+i], 0
                jmp     short loc_80483D0:

loc_80483D0:
                mov     eax, [ebp+i]

```

```

        mov     edx, 1
        mov     ebx, edx
        mov     ecx, eax
        shl     ebx, cl
        mov     eax, ebx
        and     eax, [ebp+arg_0]
        test    eax, eax
        jz      short loc_80483EB
        add     [ebp+rt], 1
loc_80483EB:
        add     [ebp+i], 1
loc_80483EF:
        cmp     [ebp+i], 1Fh
        jle     short loc_80483D0
        mov     eax, [ebp+rt]
        add     esp, 10h
        pop     ebx
        pop     ebp
        retn
f
endp

```

x64

例を64ビットに拡張するように少し変更してみましょう。

```

#include <stdio.h>
#include <stdint.h>

#define IS_SET(flag, bit)      ((flag) & (bit))

int f(uint64_t a)
{
    uint64_t i;
    int rt=0;

    for (i=0; i<64; i++)
        if (IS_SET (a, 1ULL<<i))
            rt++;

    return rt;
};

```

非最適化 GCC 4.8.2

ここまでは簡単です。

Listing 1.287: 非最適化 GCC 4.8.2

```

f:
    push     rbp
    mov     rbp, rsp

```

```

        mov     QWORD PTR [rbp-24], rdi ; a
        mov     DWORD PTR [rbp-12], 0   ; rt=0
        mov     QWORD PTR [rbp-8], 0    ; i=0
        jmp     .L2
.L4:
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
        mov     ecx, eax
; ECX = i
        shr     rdx, cl
; RDX = RDX>>CL = a>>i
        mov     rax, rdx
; RAX = RDX = a>>i
        and     eax, 1
; EAX = EAX&1 = (a>>i)&1
        test    rax, rax
; ラストビットが0か?
; そうなら、次のADD命令にスキップする
        je      .L3
        add     DWORD PTR [rbp-12], 1    ; rt++
.L3:
        add     QWORD PTR [rbp-8], 1     ; i++
.L2:
        cmp     QWORD PTR [rbp-8], 63   ; i<63?
        jbe     .L4                      ; 条件満たすなら、再度ループボディにジャン
        プする
        mov     eax, DWORD PTR [rbp-12] ; rtをリターン
        pop     rbp
        ret

```

最適化 GCC 4.8.2

Listing 1.288: 最適化 GCC 4.8.2

```

1 f:
2     xor     eax, eax                ; 変数rtはEAXレジスタに存在
3     xor     ecx, ecx                ; 変数iはECXレジスタに存在
4 .L3:
5     mov     rsi, rdi                ; 入力値をロード
6     lea     edx, [rax+1]            ; EDX=EAX+1
7 ; このEDXは rtの新たなバージョンで、
8 ; ラストビットが1の場合に、変数rtに書き込まれる
9     shr     rsi, cl                  ; RSI=RSI>>CL
10    and     esi, 1                   ; ESI=ESI&1
11 ; ラストビットが1か? そうなら、rtの新たなバージョンをEAXに書き込む
12    cmovne   eax, edx
13    add     rcx, 1                   ; RCX++
14    cmp     rcx, 64
15    jne     .L3
16    rep ret                          ; fatretの別名

```

コードは簡潔ですが、曖昧なところがあります。

これまでのすべての例では、特定のビットを比較した後に「rt」値をインクリメントしていましたが、ここでコードは「rt」を先に増やして（6行目）、新しい値をレジスタ EDX に書き込みます。したがって、最後のビットが1である場合、CMOVNE¹⁴¹ 命令（CMOVNZ¹⁴² と同義）は、EDX（「提案されたrt値」）を EAX（最後にリターンされる「現在のrt」）に戻すことによって新しい値「rt」をコミットします。

したがって、ループの各ステップで、言い換えると64回入力値に関係なく、インクリメントが実行されます。

このコードの利点は、2つのジャンプ（ループの最後で「rt」値のインクリメントをスキップする）ではなく、条件ジャンプを1つだけ（ループの最後に）含むことです。そして、それは分岐予測を持つ現代のCPUでより速く動作するでしょう：?? on page ??

最後の命令は、MSVCによって FATRET と呼ばれる REP RET (オペコード F3 C3) です。これは、RET のいくらか最適化されたバージョンであり、RET が条件ジャンプの直後にある場合、AMDは関数の最後に置くことを推奨しています：[*Software Optimization Guide for AMD Family 16h Processors*, (2013)p.15]¹⁴³.

最適化 MSVC 2010

Listing 1.289: 最適化 MSVC 2010

```
a$ = 8
f      PROC
; RCX = input value
      xor     eax, eax
      mov     edx, 1
      lea     r8d, QWORD PTR [rax+64]
; R8D=64
      npad    5
$LL4@f:
      test    rdx, rcx
; 入力値にそんな値は存在しない?
; それなら次のINC命令にスキップする
      je      SHORT $LN3@f
      inc     eax      ; rt++
$LN3@f:
      rol     rdx, 1    ; RDX=RDX<<1
      dec     r8        ; R8--
      jne     SHORT $LL4@f
      fatret   0
f      ENDP
```

ここでは、SHL の代わりに ROL 命令が使用されています。実際には、「shift left」ではなく「rotate left」ですが、この例では SHL と同じように動作します。

ローテート命令の詳細については、こちらをご覧ください：?? on page ??

¹⁴¹Conditional MOVE if Not Equal

¹⁴²Conditional MOVE if Not Zero

¹⁴³詳細はこちら: <http://repzret.org/p/repzret/>

この R8 は64から0まで数えています。*i* を逆にしたようなものです。

実行中のレジスタのテーブルを以下に示します。

RDX	R8
0x0000000000000001	64
0x0000000000000002	63
0x0000000000000004	62
0x0000000000000008	61
...	...
0x4000000000000000	2
0x8000000000000000	1

最後に、FATRET 命令がありますが、それは[1.21.5 on the preceding page](#)で説明します。

最適化 MSVC 2012

Listing 1.290: 最適化 MSVC 2012

```

a$ = 8
f    PROC
; RCX = input value
    xor     eax, eax
    mov     edx, 1
    lea     r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
    npad    5
$LL4@f:
; 1を渡す -----
    test    rdx, rcx
    je      SHORT $LN3@f
    inc     eax      ; rt++
$LN3@f:
    rol     rdx, 1   ; RDX=RDX<<1
; -----
; 2を渡す -----
    test    rdx, rcx
    je      SHORT $LN11@f
    inc     eax      ; rt++
$LN11@f:
    rol     rdx, 1   ; RDX=RDX<<1
; -----
    dec     r8      ; R8--
    jne     SHORT $LL4@f
    fatret    0
f    ENDP

```

最適化 MSVC 2012 は最適化されたMSVC 2010とほとんど同じことをしますが、どうい
うわけか、2つの同じループボディを生成して、ループカウントが64ではなく32です。

正直なところ、なぜかはわかりません。何か最適化のトリックでしょうか。ループボディ
をもう少し長くしたほうがよいのかもしれませんが。

とにかく、コンパイラの出力が本当に奇妙で非論理的なことがあります。このようなコードは完全に動作することを示すためにあげました。

ARM + 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

Listing 1.291: 最適化 Xcode 4.6.3 (LLVM) (ARMモード)

	MOV	R1, R0	
	MOV	R0, #0	
	MOV	R2, #1	
	MOV	R3, R0	
loc_2E54			
セット	TST	R1, R2, LSL R3 ; R1 & (R2<<R3) に従ってフラグを	
	ADD	R3, R3, #1 ; R3++	
R0++	ADDNE	R0, R0, #1 ; ZFフラグがTSTでクリアされた場合、	
	CMP	R3, #32	
	BNE	loc_2E54	
	BX	LR	

TST はx86では TEST と同じです。

前述のように (?? on page ??)、ARMモードでは個別のシフト命令はありません。ただし、MOV、TST、CMP、ADD、SUB、RSBなどの命令には、LSL (*Logical Shift Left*)、LSR (*Logical Shift Right*)、ASR (*Arithmetic Shift Right*)、ROR (*Rotate Right*)、RRX (*Rotate Right with Extend*) があります。

これらの変更子は、第2オペランドのシフト方法とビット数を定義します。

したがって、「TST R1, R2, LSL R3」命令はここでは $R1 \wedge (R2 \ll R3)$ として機能します。

ARM + 最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

ほぼ同じですが、Thumbモードでは LSL 修飾子を直接 TST に定義することはできないため、1つの TST の代わりに2つの LSL.W/TST 命令が使用されます。

	MOV	R1, R0	
	MOVS	R0, #0	
	MOV.W	R9, #1	
	MOVS	R3, #0	
loc_2F7A			
	LSL.W	R2, R9, R3	
	TST	R2, R1	
	ADD.W	R3, R3, #1	
	IT NE		
	ADDNE	R0, #1	
	CMP	R3, #32	
	BNE	loc_2F7A	
	BX	LR	

ARM64 + 最適化 GCC 4.9

すでに使用した64ビットの例を見てみましょう：[1.21.5 on page 400](#)

Listing 1.292: 最適化 GCC (Linaro) 4.8

```
f:
    mov     w2, 0           ; rt=0
    mov     x5, 1
    mov     w1, w2
.L2:
    lsl     x4, x5, x1      ; w4 = w5<<w1 = 1<<i
    add     w3, w2, 1      ; new_rt=rt+1
    tst     x4, x0          ; (1<<i) & a
    add     w1, w1, 1      ; i++
; TSTの結果は非ゼロか?
; そうならw2=w3 または rt=new_rt.
; そうでなければ w2=w2 または rt=rt (アイドル演算)
    csel    w2, w3, w2, ne
    cmp     w1, 64         ; i<64?
    bne     .L2            ; はい
    mov     w0, w2         ; rtをリターン
    ret
```

結果は、GCCがx64に対して生成するものと非常によく似ています：[1.288 on page 401](#)

CSEL 命令は「Conditional SElect」です。TST で設定されたフラグに応じて1つの変数が2つだけ選択され、値が「rt」変数を保持する W2 にコピーされます。

ARM64 + 非最適化 GCC 4.9

もう一度、すでに使用した64ビットの例について作業します：[1.21.5 on page 400](#) 例によって、コードはより冗長です。

Listing 1.293: 非最適化 GCC (Linaro) 4.8

```
f:
    sub     sp, sp, #32
    str     x0, [sp,8]      ; "a" 値をレジスタの保存エリアに保存
    str     wzr, [sp,24]    ; rt=0
    str     wzr, [sp,28]    ; i=0
    b       .L2
.L4:
    ldr     w0, [sp,28]
    mov     x1, 1
    lsl     x0, x1, x0      ; X0 = X1<<X0 = 1<<i
    mov     x1, x0
; X1 = 1<<i
    ldr     x0, [sp,8]
; X0 = a
    and     x0, x1, x0
; X0 = X1&X0 = (1<<i) & a
; X0はゼロを含むか? そうなら、.L3にジャンプし、"rt" インクリメントをスキップ
    cmp     x0, xzr
```

```

        beq      .L3
; rt++
        ldr      w0, [sp,24]
        add      w0, w0, 1
        str      w0, [sp,24]
.L3:
; i++
        ldr      w0, [sp,28]
        add      w0, w0, 1
        str      w0, [sp,28]
.L2:
; i<=63か? そうなら.L4にジャンプ
        ldr      w0, [sp,28]
        cmp      w0, 63
        ble      .L4
; rtをリターン
        ldr      w0, [sp,24]
        add      sp, sp, 32
        ret

```

MIPS

非最適化 GCC

Listing 1.294: 非最適化 GCC 4.4.5 (IDA)

```

f:
; IDAは変数名を知らないので、手動で与えます
rt      = -0x10
i       = -0xC
var_4   = -4
a       = 0

        addiu    $sp, -0x18
        sw       $fp, 0x18+var_4($sp)
        move     $fp, $sp
        sw       $a0, 0x18+a($fp)
; rtを初期化し、i変数を0にします
        sw       $zero, 0x18+rt($fp)
        sw       $zero, 0x18+i($fp)
; ループチェック命令にジャンプ
        b        loc_68
        or        $at, $zero ; 分岐遅延スロット, NOP

loc_20:
        li       $v1, 1
        lw       $v0, 0x18+i($fp)
        or        $at, $zero ; 遅延スロットをロード, NOP
        sllv     $v0, $v1, $v0
; $v0 = 1<<i
        move     $v1, $v0
        lw       $v0, 0x18+a($fp)

```

```

        or      $at, $zero ; 遅延スロットをロード, NOP
        and     $v0, $v1, $v0
; $v0 = a & (1<<i)
; a & (1<<i) はゼロと等しいか? 等しければloc_58へ
        beqz    $v0, loc_58
        or      $at, $zero
; no jump occurred, that means a & (1<<i)!=0, so increment "rt" then:
        lw      $v0, 0x18+rt($fp)
        or      $at, $zero ; 遅延スロットをロード, NOP
        addiu   $v0, 1
        sw      $v0, 0x18+rt($fp)

loc_58:
; iをインクリメント:
        lw      $v0, 0x18+i($fp)
        or      $at, $zero ; 遅延スロットをロード, NOP
        addiu   $v0, 1
        sw      $v0, 0x18+i($fp)

loc_68:
; load iをロードし0x20 (32) と比較
; 0x20 (32) 未満の場合loc_20にジャンプ:
        lw      $v0, 0x18+i($fp)
        or      $at, $zero ; 遅延スロットをロード, NOP
        slti    $v0, 0x20 # ' '
        bnez    $v0, loc_20
        or      $at, $zero ; 分岐遅延スロット, NOP
; 関数エピローグ。rtをリターン
        lw      $v0, 0x18+rt($fp)
        move    $sp, $fp ; 遅延スロットをロード
        lw      $fp, 0x18+var_4($sp)
        addiu   $sp, 0x18 ; 遅延スロットをロード
        jr      $ra
        or      $at, $zero ; 分岐遅延スロット, NOP

```

これは冗長です。ローカル変数はすべてローカルスタックに配置され、必要なとき毎にリロードされます。

SLLV 命令は「Shift Word Left Logical Variable」で、SLL との違いは SLL 命令にエンコードされたシフトの量だけです。（そして結果として固定されています）しかし、SLLV はレジスタからシフト量を取ってきます。

最適化 GCC

これはより簡潔です。1つではなく2つシフト命令がありますが、なぜでしょうか？

最初の SLLV 命令を 2つめの SLLV にジャンプする無条件分岐命令に置き換えることは可能です。しかし、これは関数内の別の分岐命令であり、常にそれらを取り除くのは好都合です：?? on page ??

Listing 1.295: 最適化 GCC 4.4.5 (IDA)

```

f:
; $a0=a
; 変数rtは $v0に存在する
        move    $v0, $zero
; 変数iは $v1に存在する
        move    $v1, $zero
        li      $t0, 1
        li      $a3, 32
        sllv    $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i

loc_14:
        and     $a1, $a0
; $a1 = a&(1<<i)
; iをインクリメント
        addiu   $v1, 1
; a&(1<<i)==0ならloc_28にジャンプしrtをインクリメント
        beqz    $a1, loc_28
        addiu   $a2, $v0, 1
; BEQZが実行されなければ、更新したrtを $v0に保存
        move    $v0, $a2

loc_28:
; i!=32なら、loc_14にジャンプし、次のシフトした値を準備
        bne     $v1, $a3, loc_14
        sllv    $a1, $t0, $v1
; リターン
        jr      $ra
        or      $at, $zero ; 分岐遅延スロット、NOP

```

第1.21.6節結論

C/C++ のシフト演算子 << および >> と同様に、x86のシフト命令は SHR/SHL（符号なしの値）と SAR/SHL（符号付きの値）です。

ARMのシフト命令は、LSR/LSL（符号なし値の場合）と ASR/LSL（符号付き値の場合）です。

シフトサフィックスをいくつかの命令（「データ処理命令」と呼ばれます）に追加することもできます。

特定のビットをチェックする（コンパイル段階で知られている）

0b10000000ビット（0x40）がレジスタの値に存在するかどうかをテストします。

Listing 1.296: C/C++

```

if (input&0x40)
    ...

```

Listing 1.297: x86

```
TEST REG, 40h
JNZ is_set
; ビットはセットされていない
```

Listing 1.298: x86

```
TEST REG, 40h
JZ is_cleared
; ビットはセット
```

Listing 1.299: ARM (ARMモード)

```
TST REG, #0x40
BNE is_set
; ビットはセットされていない
```

場合によっては、TEST の代わりに AND が使用されますが、設定されるフラグは同じです。

特定のビットをチェックする（実行時に指定する）

これは通常、この C/C++ コードスニペットによって行われます（ n ビット右にシフトし、次に最下位ビットをカットします）。

Listing 1.300: C/C++

```
if ((value>>n)&1)
    ....
```

これは通常、x86コードで次のように実装されています。

Listing 1.301: x86

```
; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1
```

または（1ビット左シフトを n 回、入力値でこのビットを分離し、ゼロでないかどうかをチェックする）

Listing 1.302: C/C++

```
if (value & (1<<n))
    ....
```

これは通常、x86コードで次のように実装されています。

Listing 1.303: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

特定のビットを設定する（コンパイル時に知られている）

Listing 1.304: C/C++

```
value=value|0x40;
```

Listing 1.305: x86

```
OR REG, 40h
```

Listing 1.306: ARM (ARMモード) and ARM64

```
ORR R0, R0, #0x40
```

特定のビットを設定する（実行時に指定する）

Listing 1.307: C/C++

```
value=value|(1<<n);
```

これは通常、x86コードで次のように実装されています。

Listing 1.308: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

明確な特定のビット（コンパイル段階で知られている）

逆の値で AND 演算を適用するだけです：

Listing 1.309: C/C++

```
value=value&(~0x40);
```

Listing 1.310: x86

```
AND REG, 0FFFFFFBFh
```

Listing 1.311: x64

```
AND REG, 0FFFFFFFFFFFFFFBFh
```

これは、実際には、1を除いてすべてのビットを設定しています。

ARMモードのARMには BIC 命令があります。これは NOT +AND 命令ペアのように動作します。

Listing 1.312: ARM (ARMモード)

```
BIC R0, R0, #0x40
```

特定のビットをクリア（実行時に指定）

Listing 1.313: C/C++

```
value=value&~(1<<n));
```

Listing 1.314: x86

```
; CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

第1.21.7節練習問題

- <http://challenges.re/67>
- <http://challenges.re/68>
- <http://challenges.re/69>
- <http://challenges.re/70>

第1.22節擬似乱数生成器としての線形合同生成器

おそらく、線形合同ジェネレータは、乱数を生成するための最も簡単な方法です。

今日では¹⁴⁴選択されませんが、とても単純です（1回の乗算、1回の加算とAND演算）。これを例として使用できます。

```
#include <stdint.h>

// ニューメリカルレシピ本からとった定数
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}
```

¹⁴⁴メルセンヌツイスターの方がいいです

2つの関数があります：最初のは内部状態を初期化するために使用され、2つ目は擬似乱数を生成するために呼び出されます。

アルゴリズムでは2つの定数が使用されていることがわかります。それらは [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)] から取られています。

#define C/C++ 命令文を使ってそれらを定義しましょう。これはマクロです。

C/C++ マクロと定数の違いは、すべてのマクロが C/C++ プリプロセッサでその値に置換され、変数と異なりメモリを使用しないことです。

対照的に、定数は読み取り専用変数です。

定数変数のポインタ（またはアドレス）を取ることは可能ですが、マクロではできません。

C標準の `my_rand()` は0から32767の範囲の値を返さなければならないため、最後のAND演算が必要です。

32ビットの擬似乱数値を取得する場合は、最後のAND演算を省略してください。

第1.22.1節x86

Listing 1.315: 最適化 MSVC 2013

```

_BSS    SEGMENT
_rand_state DD    01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
        mov     eax, DWORD PTR _init$[esp-4]
        mov     DWORD PTR _rand_state, eax
        ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
        imul    eax, DWORD PTR _rand_state, 1664525
        add     eax, 1013904223 ; 3c6ef35fH
        mov     DWORD PTR _rand_state, eax
        and     eax, 32767      ; 00007fffH
        ret     0
_rand   ENDP
_TEXT   ENDS

```

ここでは、両方の定数がコードに埋め込まれています。割り当てられたメモリはありません。

`my_srand()` 関数は入力値を内部の `rand_state` 変数にコピーするだけです。

`my_rand()` はそれを受け取り、次の `rand_state` を計算し、それを切り取り、EAXレジスタに残します。

最適化されていないバージョンはより冗長です。

Listing 1.316: 非最適化 MSVC 2013

```

_BSS    SEGMENT
_rand_state DD    01H DUP (?)
_BSS    ENDS

_init$ = 8
_srand  PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _init$[ebp]
    mov     DWORD PTR _rand_state, eax
    pop     ebp
    ret     0
_srand  ENDP

_TEXT   SEGMENT
_rand   PROC
    push    ebp
    mov     ebp, esp
    imul    eax, DWORD PTR _rand_state, 1664525
    mov     DWORD PTR _rand_state, eax
    mov     ecx, DWORD PTR _rand_state
    add     ecx, 1013904223 ; 3c6ef35fH
    mov     DWORD PTR _rand_state, ecx
    mov     eax, DWORD PTR _rand_state
    and     eax, 32767      ; 00007fffH
    pop     ebp
    ret     0
_rand   ENDP

_TEXT   ENDS

```

第1.22.2節x64

x64のバージョンはほとんど同じで、64ビットではなく32ビットのレジスタを使用しています。（ここで *int* 値を使用しているためです）

しかし、`my_srand()` は入力引数をスタックからではなく ECX レジスタから取ります：

Listing 1.317: 最適化 MSVC 2013 x64

```

_BSS    SEGMENT
rand_state DD    01H DUP (?)
_BSS    ENDS

init$ = 8
my_srand PROC
; ECX = 入力引数
    mov     DWORD PTR rand_state, ecx
    ret     0
my_srand ENDP

_TEXT   SEGMENT

```

```

my_rand PROC
    imul    eax, DWORD PTR rand_state, 1664525 ; 0019660dH
    add     eax, 1013904223 ; 3c6ef35fH
    mov     DWORD PTR rand_state, eax
    and     eax, 32767      ; 00007fffH
    ret     0
my_rand ENDP

_TEXT     ENDS

```

GCCコンパイラはほとんど同じコードを生成します。

第1.22.3節32ビットARM

Listing 1.318: 最適化 Keil 6/2013 (ARMモード)

```

my_srand PROC
    LDR     r1, |L0.52| ; ポインタをrand_stateにロード
    STR     r0, [r1, #0] ; rand_stateを保存
    BX      lr
ENDP

my_rand PROC
    LDR     r0, |L0.52| ; ポインタをrand_stateにロード
    LDR     r2, |L0.56| ; RNG_aをロード
    LDR     r1, [r0, #0] ; rand_stateをロード
    MUL     r1, r2, r1
    LDR     r2, |L0.60| ; RNG_cをロード
    ADD     r1, r1, r2
    STR     r1, [r0, #0] ; rand_stateを保存
; AND with 0x7FFF:
    LSL     r0, r1, #17
    LSR     r0, r0, #17
    BX      lr
ENDP

|L0.52|
DCD       ||.data||
|L0.56|
DCD       0x0019660d
|L0.60|
DCD       0x3c6ef35f

AREA ||.data||, DATA, ALIGN=2

rand_state
DCD       0x00000000

```

32ビット定数をARM命令に埋め込むことはできないため、Keilはそれらを外部に配置して追加する必要があります。興味深いことに、0x7FFF定数も埋め込むことはできません。Keilがやっているのは、`rand_state` を17ビット左にシフトし、右に17ビットシフトすることです。これは、C/C++ の $(rand_state \ll 17) \gg 17$ 命令文に似ています。それは役に立

たない操作だと思われますが、それは17ビットをクリアして15ビットをそのままにして、これが結局のところ私たちの目標です。

Thumbモードの最適化 Keil はほとんど同じコードが生成します。

第1.22.4節MIPS

Listing 1.319: 最適化 GCC 4.4.5 (IDA)

```
my_srand:
; $a0にrand_stateを保存
    lui    $v0, (rand_state >> 16)
    jr     $ra
    sw     $a0, rand_state
my_rand:
; rand_stateを $v0にロード
    lui    $v1, (rand_state >> 16)
    lw     $v0, rand_state
    or     $at, $zero ; ロード遅延スロット
; rand_stateに1554525 (RNG_a) を乗算した結果を $v0に
    sll    $a1, $v0, 2
    sll    $a0, $v0, 4
    addu   $a0, $a1, $a0
    sll    $a1, $a0, 6
    subu   $a0, $a1, $a0
    addu   $a0, $v0
    sll    $a1, $a0, 5
    addu   $a0, $a1
    sll    $a0, $a0, 3
    addu   $v0, $a0, $v0
    sll    $a0, $v0, 2
    addu   $v0, $a0
; 1013904223 (RNG_c) を加算
; LI命令はIDAがLUIとORIを合体したもの
    li     $a0, 0x3C6EF35F
    addu   $v0, $a0
; rand_stateを保存
    sw     $v0, (rand_state & 0xFFFF)($v1)
    jr     $ra
    andi   $v0, 0x7FFF ; 分岐遅延スロット
```

おっと、ここでは1つの定数 (0x3C6EF35Fまたは1013904223) しか表示されません。もう1つはどこでしょうか (1664525)？

1664525による乗算は、シフトと加算だけを使用して実行されるようです！この仮定を確認してみましょう：

```
#define RNG_a 1664525

int f (int a)
{
    return a*RNG_a;
}
```

Listing 1.320: 最適化 GCC 4.4.5 (IDA)

```
f:
        sll      $v1, $a0, 2
        sll      $v0, $a0, 4
        addu     $v0, $v1, $v0
        sll      $v1, $v0, 6
        subu     $v0, $v1, $v0
        addu     $v0, $a0
        sll      $v1, $v0, 5
        addu     $v0, $v1
        sll      $v0, 3
        addu     $a0, $v0, $a0
        sll      $v0, $a0, 2
        jr       $ra
        addu     $v0, $a0, $v0 ; branch delay slot
```

本当に！

MIPSの再配置

また、メモリやストアから実際にメモリにロードする操作がどのように機能するかにも焦点を当てます。

このリストはIDAによって作成され、IDAはいくつかの詳細を隠しています。

objdumpを2回実行します：逆アセンブルされたリストと再配置リストを取得します。

Listing 1.321: 最適化 GCC 4.4.5 (objdump)

```
# objdump -D rand_03.o

...

00000000 <my_srand>:
   0: 3c020000      lui      v0,0x0
   4: 03e00008      jr       ra
   8: ac440000      sw       a0,0(v0)

0000000c <my_rand>:
   c: 3c030000      lui      v1,0x0
  10: 8c620000      lw       v0,0(v1)
  14: 00200825      move     at,at
  18: 00022880      sll      a1,v0,0x2
  1c: 00022100      sll      a0,v0,0x4
  20: 00a42021      addu     a0,a1,a0
  24: 00042980      sll      a1,a0,0x6
  28: 00a42023      subu     a0,a1,a0
  2c: 00822021      addu     a0,a0,v0
  30: 00042940      sll      a1,a0,0x5
  34: 00852021      addu     a0,a0,a1
  38: 000420c0      sll      a0,a0,0x3
  3c: 00821021      addu     v0,a0,v0
  40: 00022080      sll      a0,v0,0x2
  44: 00441021      addu     v0,v0,a0
```

```

48: 3c043c6e    lui    a0,0x3c6e
4c: 3484f35f    ori    a0,a0,0xf35f
50: 00441021    addu   v0,v0,a0
54: ac620000    sw     v0,0(v1)
58: 03e00008    jr     ra
5c: 30427fff    andi   v0,v0,0x7fff

```

...

```
# objdump -r rand_03.o
```

...

```

RELOCATION RECORDS FOR [.text]:
OFFSET    TYPE             VALUE
00000000  R_MIPS_HI16           .bss
00000008  R_MIPS_L016           .bss
0000000c  R_MIPS_HI16           .bss
00000010  R_MIPS_L016           .bss
00000054  R_MIPS_L016           .bss

```

...

my_srand() 関数の2つの再配置を考えてみましょう。

最初のアドレス0は R_MIPS_HI16 のタイプを持ち、アドレス8の2番目のアドレスは R_MIPS_L016 のタイプです。

つまり、.bssセグメントの先頭のアドレスは、0（アドレスの上位部分）および8（アドレスの下位部分）のアドレスに書き込まれることを意味します。

rand_state 変数は、.bssセグメントの先頭にあります。

したがって、命令 LUI と SW のオペランドにはゼロがあります。何もまだ存在しないからです。コンパイラは何をそこに書き込んだらいいかわかりません。

リンカがこれを修正し、アドレスの上位部分が LUI のオペランドに書き込まれ、アドレスの下位部分が SW のオペランドに書き込まれます。

SW はアドレスの下位部分とレジスタ \$V0 にあるものを合計します（上位部分はそこにあります）。

これは my_rand() 関数の場合と同じです。R_MIPS_HI16 再配置は、リンカに.bssセグメントアドレスの上位部分を LUI 命令に書き込むように指示します。

したがって、rand_state 変数アドレスの上位部分はレジスタ \$V1 に存在します。

アドレス0x10にある LW 命令は、上位部分と下位部分を合計し、rand_state 変数の値を \$V0 にロードします。

アドレス0x54にある SW 命令は、加算を再度行い、新しい値をrand_stateグローバル変数に格納します。

IDAは、ロード中に再配置を処理するため、これらの詳細は隠していますが、それらを念頭に置いておく必要があります。

第1.22.5節スレッドセーフ版の例

この例のスレッドセーフ版は、後で説明します：[?? on page ??](#)

第1.23節構造体

C/C++ 構造体はいくつかの前提があり、単なる変数の集合であり、常に一緒にメモリに格納され、同じ型の必要はありません ¹⁴⁵

第1.23.1節MSVC: SYSTEMTIME example

時間を表現する SYSTEMTIME¹⁴⁶ win32構造体を取りあげましょう。

このように定義されます。

Listing 1.322: WinBase.h

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

現在時刻を取得するCの関数を書いてみましょう。

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

次の結果を得ます。(MSVC 2010)

Listing 1.323: MSVC 2010 /GS-

```
_t$ = -16 ; size = 16
```

¹⁴⁵[AKA 「異種コンテナです」](#)

¹⁴⁶[MSDN: SYSTEMTIME structure](#)

```

_main    PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    lea     eax, DWORD PTR _t$[ebp]
    push    eax
    call    DWORD PTR __imp__GetSystemTime@4
    movzx   ecx, WORD PTR _t$[ebp+12] ; wSecond
    push    ecx
    movzx   edx, WORD PTR _t$[ebp+10] ; wMinute
    push    edx
    movzx   eax, WORD PTR _t$[ebp+8] ; wHour
    push    eax
    movzx   ecx, WORD PTR _t$[ebp+6] ; wDay
    push    ecx
    movzx   edx, WORD PTR _t$[ebp+2] ; wMonth
    push    edx
    movzx   eax, WORD PTR _t$[ebp] ; wYear
    push    eax
    push    OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
    call    _printf
    add     esp, 28
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main    ENDP

```

16バイトがローカルスタック上に構造体のために確保されていて、これはちょうど `sizeof(WORD)*8` です。(構造体にあるWORD変数8つ分です)

構造体は `wYear` フィールドから始まるという事実に注意してください。SYSTEMTIME構造体へのポインタが `GetSystemTime()`¹⁴⁷ に渡されますが、`wYear` フィールドへのポインタが渡されているとも言えます。そしてこれは同じです！`GetSystemTime()` は現在の年をWORDポインタが示すところに書き込み、それから2バイトを前方にシフトし、現在の月を書き込み、などなど。

¹⁴⁷[MSDN: SYSTEMTIME structure](#)

OllyDbg

この例を/GS- /MD オプション付きでMSVC 2010でコンパイルし OllyDbg で実行してみましょう。

データのウィンドウを開き、GetSystemTime() 関数の最初の引数として渡されたアドレスにスタックし、実行されるまで待機しましょう。このようになります。

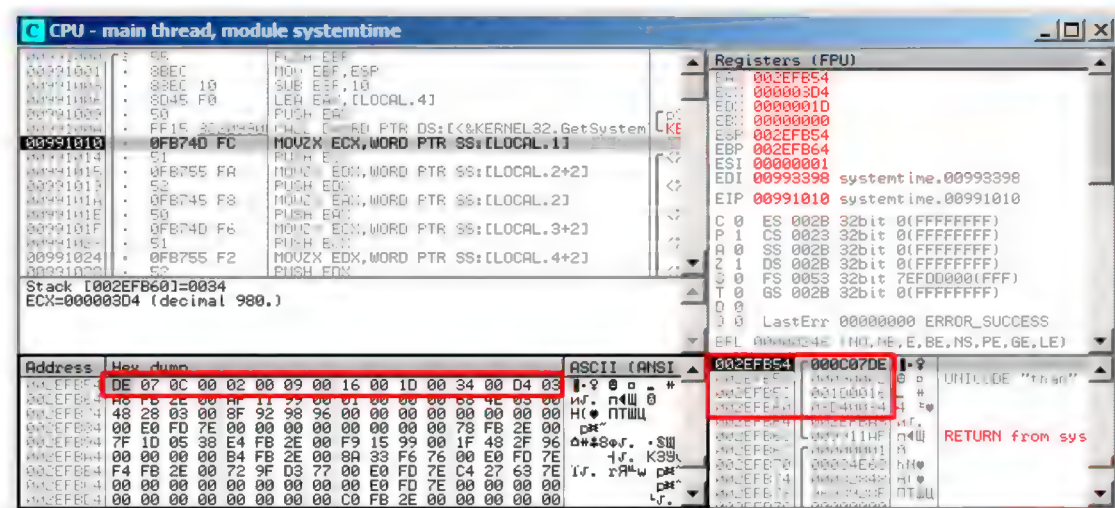


図 1.104: OllyDbg: GetSystemTime() が実行された

私のコンピュータ上での関数のシステム時間は2014年12月9日、22時29分52秒です。

Listing 1.324: printf() output

```
2014-12-09 22:29:52
```

このような16バイトをデータウィンドウにみることができます。

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 D4 03
```

各2バイトが構造体のフィールドを表します。endiannessはリトルエンディアンなので、低位バイトが最初に見え、高位バイトがその後です。

したがって、これらは現在メモリに格納されている値です。

16進数	10進数	フィールド名
0x07DE	2014	wYear
0x000C	12	wMonth
0x0002	2	wDayOfWeek
0x0009	9	wDay
0x0016	22	wHour
0x001D	29	wMinute
0x0034	52	wSecond
0x03D4	980	wMilliseconds

同じ値がスタックウィンドウに表示されますが、32ビットの値としてグループ分けされています。

そして、printf() は必要な値だけを取り出してコンソールに出力します。

いくつかの値は printf() (wDayOfWeek と wMilliseconds) によって出力されませんが、使用可能なメモリ上にあります。

構造体を配列で置き換える

構造体のフィールドは単に隣り合った変数で、以下のようにすることで簡単にデモンストレーションできます。SYSTEMTIME 構造体の表現を覚えておいて、この簡単な例をこのように書き換えることが可能です。

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /* wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] /* wSecond */
        ↵ /* */);

    return;
};
```

コンパイラは少し不満を言います。

```
systemtime2.c(7) : warning C4133: 'function' : incompatible types - from '↵
    ↵ WORD [8]' to 'LPSYSTEMTIME'
```

とはいえ、このようなコードを生成します。

Listing 1.325: 非最適化 MSVC 2010

```
$SG78573 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16    ; size = 16
_main PROC
    push     ebp
    mov     ebp, esp
    sub     esp, 16
    lea     eax, DWORD PTR _array$[ebp]
    push     eax
    call    DWORD PTR __imp__GetSystemTime@4
    movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
    push     ecx
    movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
    push     edx
    movzx   eax, WORD PTR _array$[ebp+8] ; wHour
```

```

        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573 ; '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main     ENDP

```

そして同じように機能します！

アセンブリ形式の結果が前のコンパイルの結果と区別できないことは非常に興味深いことです。

だから、このコードを見て、構造体が宣言されているか、配列なのかははっきりとすることができません。

とはいえ、普通の人は都合がよいわけではないのでこういうことはしません。

また、構造体のフィールドは開発者によって変更されたり、スワップされたりすることがあります。

この例は、構造体の場合とまったく同じであるため、OllyDbg ではこの例を学習しません。

第1.23.2節 **malloc()** を使って構造体のための領域を割り当てよう

場合によっては、ローカルスタックではなく[ヒープ](#)内に構造体を配置する方が簡単な場合があります。

```

#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
}

```

```
};
```

最適化（/Ox）でコンパイルして、必要なものを見るのは簡単です。

Listing 1.326: 最適化 MSVC

```
_main PROC
    push    esi
    push    16
    call    _malloc
    add     esp, 4
    mov     esi, eax
    push    esi
    call    DWORD PTR __imp__GetSystemTime@4
    movzx   eax, WORD PTR [esi+12] ; wSecond
    movzx   ecx, WORD PTR [esi+10] ; wMinute
    movzx   edx, WORD PTR [esi+8]  ; wHour
    push    eax
    movzx   eax, WORD PTR [esi+6] ; wDay
    push    ecx
    movzx   ecx, WORD PTR [esi+2] ; wMonth
    push    edx
    movzx   edx, WORD PTR [esi]   ; wYear
    push    eax
    push    ecx
    push    edx
    push    OFFSET $SG78833
    call    _printf
    push    esi
    call    _free
    add     esp, 32
    xor     eax, eax
    pop     esi
    ret     0
_main ENDP
```

したがって、`sizeof(SYSTEMTIME) = 16`であり、これは`malloc()`によって割り当てられる正確なバイト数です。EAX レジスタ内の新しく割り当てられたメモリブロックへのポインタを返し、ESI レジスタに移動します。`GetSystemTime()` win32関数は ESI の値を保存するため、ここで保存されず、`GetSystemTime()` 呼び出しの後も引き続き使用されます。

新しい命令 —MOVZX (*Move with Zero eXtend*)。ほとんどの場合、MOVZX として使用できますが、残りのビットは0に設定されます。これは `printf()` が32ビットの *int* を必要とするためですが、構造体にWORDがあるためです。つまり、16ビットの符号なし型です。そのため、WORDの値を *int* にコピーすることにより、16から31までのビットをクリアする必要があります。ランダムノイズが存在する可能性があります。これはレジスタの前の操作から残されているためです。

この例では、構造を8つのWORDの配列として表すことができます。

```
#include <windows.h>
#include <stdio.h>
```

```

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond */);

    free (t);

    return;
};

```

次の結果を得ます。

Listing 1.327: 最適化 MSVC

```

$SG78594 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main  PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main  ENDP

```

ここでも、前のコードと区別できないコードがあります。

また、実際にあなたが何をしているのか分からない限り、実際にはこれをしていないことに注意しなければなりません。

第1.23.3節UNIX: struct tm

Linux

Linuxの time.h の tm 構造体を例にとりましょう。

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

GCC 4.4.1でコンパイルしましょう。

Listing 1.328: GCC 4.4.1

```
main proc near
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; time() への第一引数
    call    time
    mov     [esp+3Ch], eax
    lea     eax, [esp+3Ch] ; take pointer to what time() returned
    lea     edx, [esp+10h] ; at ESP+10h struct tm will begin
    mov     [esp+4], edx ; 構造体へのポインタを渡す
    mov     [esp], eax ; time() の結果へのポインタを渡す
    call    localtime_r
    mov     eax, [esp+24h] ; tm_year
    lea     edx, [eax+76Ch] ; edx=eax+1900
    mov     eax, offset format ; "Year: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+20h] ; tm_mon
    mov     eax, offset aMonthD ; "Month: %d\n"
```

```

mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+1Ch]      ; tm_mday
mov     eax, offset aDayD   ; "Day: %d\n"
mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+18h]      ; tm_hour
mov     eax, offset aHourD ; "Hour: %d\n"
mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+14h]      ; tm_min
mov     eax, offset aMinutesD ; "Minutes: %d\n"
mov     [esp+4], edx
mov     [esp], eax
call    printf
mov     edx, [esp+10h]
mov     eax, offset aSecondsD ; "Seconds: %d\n"
mov     [esp+4], edx      ; tm_sec
mov     [esp], eax
call    printf
leave
retn
main endp

```

どういうわけか、IDA はローカルスタックにローカル変数を書き込みませんでした。しかし、経験を積んだリバース・エンジニアなので:-) この単純な例では情報なしで実行できるかもしれません。

lea edx, [eax+76Ch] にも注意してください。この命令は 0x76C (1900) を EAX の値に追加するだけですが、フラグは変更しません。LEA (?? on page ??) に関する関連セクションも参照してください。

GDB

例をGDBにロードしてみましょう。 [148](#)

Listing 1.329: GDB

```

dennis@ubuntuvirtual-machine:~/polygon$ date
Mon Jun  2 18:10:37 EEST 2014
dennis@ubuntuvirtual-machine:~/polygon$ gcc GCC_tm.c -o GCC_tm
dennis@ubuntuvirtual-machine:~/polygon$ gdb GCC_tm
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/GCC_tm...(no debugging symbols found)
(gdb) b printf

```

¹⁴⁸ デモの目的で、date の結果が少し修正されます。もちろん、同じ秒で、GDBをすばやく実行することはできません。

```

Breakpoint 1 at 0x08048330
(gdb) run
Starting program: /home/dennis/polygon/GCC_tm

Breakpoint 1, __printf (format=0x080485c0 "Year: %d\n") at printf.c:29
29      printf.c: No such file or directory.
(gdb) x/20x $esp
0xbffff0dc: 0x080484c3      0x080485c0      0x0000007de      0x00000000
0xbffff0ec: 0x08048301      0x538c93ed      0x000000025      0x0000000a
0xbffff0fc: 0x000000012      0x000000002      0x000000005      0x000000072
0xbffff10c: 0x000000001      0x000000098      0x000000001      0x000002a30
0xbffff11c: 0x0804b090      0x08048530      0x000000000      0x00000000
(gdb)

```

私たちは簡単にスタック内の構造を見つけることができます。まず、*time.h* でどのように定義されているのかを見てみましょう。

Listing 1.330: time.h

```

struct tm
{
    int    tm_sec;
    int    tm_min;
    int    tm_hour;
    int    tm_mday;
    int    tm_mon;
    int    tm_year;
    int    tm_wday;
    int    tm_yday;
    int    tm_isdst;
};

```

ここでは、SYSTEMTIMEのWORDの代わりに32ビット *int* が使用されていることに注意してください。したがって、各フィールドは32ビットを占有します。

スタック内の構造体のフィールドは次のとおりです。

```

0xbffff0dc: 0x080484c3      0x080485c0      0x0000007de      0x00000000
0xbffff0ec: 0x08048301      0x538c93ed      0x000000025      0x0000000a min
0xbffff0fc: 0x000000012      hour 0x000000002      mday 0x000000005      mon 0x000000072 ↗
           ↘ year
0xbffff10c: 0x000000001      wday 0x000000098      yday 0x000000001      isdst 0x000002a30
0xbffff11c: 0x0804b090      0x08048530      0x000000000      0x00000000

```

またはテーブルとして。

16進数	10進数	フィールド名
0x00000025	37	tm_sec
0x0000000a	10	tm_min
0x00000012	18	tm_hour
0x00000002	2	tm_mday
0x00000005	5	tm_mon
0x00000072	114	tm_year
0x00000001	1	tm_wday
0x00000098	152	tm_yday
0x00000001	1	tm_isdst

SYSTEMTIMEと同じように ([1.23.1 on page 418](#))

tm_wday, tm_yday, tm_isdstなど、使用されていない他のフィールドも使用できます。

ARM

最適化 Keil 6/2013 (Thumbモード)

同じ例

Listing 1.331: 最適化 Keil 6/2013 (Thumbモード)

```

var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

    PUSH    {LR}
    MOVS    R0, #0          ; timer
    SUB     SP, SP, #0x34
    BL      time
    STR     R0, [SP, #0x38+timer]
    MOV     R1, SP          ; tp
    ADD     R0, SP, #0x38+timer ; timer
    BL      localtime_r
    LDR     R1, =0x76C
    LDR     R0, [SP, #0x38+var_24]
    ADDS    R1, R0, R1
    ADR     R0, aYearD      ; "Year: %d\n"
    BL      __2printf
    LDR     R1, [SP, #0x38+var_28]
    ADR     R0, aMonthD     ; "Month: %d\n"
    BL      __2printf
    LDR     R1, [SP, #0x38+var_2C]
    ADR     R0, aDayD       ; "Day: %d\n"
    BL      __2printf
    LDR     R1, [SP, #0x38+var_30]
    ADR     R0, aHourD      ; "Hour: %d\n"
    BL      __2printf

```



```

LDR    R1, [SP,#0x38+var_34]
ADR    R0, aMinutesD    ; "Minutes: %d\n"
BL     _2printf
LDR    R1, [SP,#0x38+var_38]
ADR    R0, aSecondsD    ; "Seconds: %d\n"
BL     _2printf
ADD    SP, SP, #0x34
POP    {PC}

```

最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

IDA は tm 構造体を「知っています」。(IDA は localtime_r() のようなライブラリ関数の引数の型を「知っている」からです)

ここでは構造要素のアクセスとその名前を示しています。

Listing 1.332: 最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

```

var_38 = -0x38
var_34 = -0x34

    PUSH {R7,LR}
    MOV  R7, SP
    SUB  SP, SP, #0x30
    MOVS R0, #0    ; time_t *
    BLX  _time
    ADD  R1, SP, #0x38+var_34 ; struct tm *
    STR  R0, [SP,#0x38+var_38]
    MOV  R0, SP    ; time_t *
    BLX  _localtime_r
    LDR  R1, [SP,#0x38+var_34.tm_year]
    MOV  R0, 0xF44 ; "Year: %d\n"
    ADD  R0, PC    ; char *
    ADDW R1, R1, #0x76C
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_mon]
    MOV  R0, 0xF3A ; "Month: %d\n"
    ADD  R0, PC    ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_mday]
    MOV  R0, 0xF35 ; "Day: %d\n"
    ADD  R0, PC    ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_hour]
    MOV  R0, 0xF2E ; "Hour: %d\n"
    ADD  R0, PC    ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34.tm_min]
    MOV  R0, 0xF28 ; "Minutes: %d\n"
    ADD  R0, PC    ; char *
    BLX  _printf
    LDR  R1, [SP,#0x38+var_34]

```

```

MOV R0, 0xF25 ; "Seconds: %d\n"
ADD R0, PC ; char *
BLX _printf
ADD SP, SP, #0x30
POP {R7,PC}

...

00000000 tm      struc ; (sizeof=0x2C, standard type)
00000000 tm_sec   DCD ?
00000004 tm_min   DCD ?
00000008 tm_hour  DCD ?
0000000C tm_mday  DCD ?
00000010 tm_mon   DCD ?
00000014 tm_year  DCD ?
00000018 tm_wday  DCD ?
0000001C tm_yday  DCD ?
00000020 tm_isdst DCD ?
00000024 tm_gmtoff DCD ?
00000028 tm_zone  DCD ? ; offset
0000002C tm      ends

```

MIPS

Listing 1.333: 最適化 GCC 4.4.5 (IDA)

```

1 main:
2
3 ; IDAは構造体のフィールド名がわからないので、手動で名前を付けます
4
5 var_40      = -0x40
6 var_38      = -0x38
7 seconds     = -0x34
8 minutes     = -0x30
9 hour        = -0x2C
10 day         = -0x28
11 month       = -0x24
12 year        = -0x20
13 var_4       = -4
14
15 lui        $gp, (__gnu_local_gp >> 16)
16 addiu      $sp, -0x50
17 la         $gp, (__gnu_local_gp & 0xFFFF)
18 sw         $ra, 0x50+var_4($sp)
19 sw         $gp, 0x50+var_40($sp)
20 lw         $t9, (time & 0xFFFF)($gp)
21 or         $at, $zero ; ロード遅延スロット, NOP
22 jalr       $t9
23 move       $a0, $zero ; 分岐遅延スロット, NOP
24 lw         $gp, 0x50+var_40($sp)
25 addiu      $a0, $sp, 0x50+var_38
26 lw         $t9, (localtime_r & 0xFFFF)($gp)
27 addiu      $a1, $sp, 0x50+seconds

```

```

28 jalr    $t9
29 sw      $v0, 0x50+var_38($sp) ; 分岐遅延スロット
30 lw      $gp, 0x50+var_40($sp)
31 lw      $a1, 0x50+year($sp)
32 lw      $t9, (printf & 0xFFFF)($gp)
33 la      $a0, $LC0          # "Year: %d\n"
34 jalr    $t9
35 addiu   $a1, 1900 ; 分岐遅延スロット
36 lw      $gp, 0x50+var_40($sp)
37 lw      $a1, 0x50+month($sp)
38 lw      $t9, (printf & 0xFFFF)($gp)
39 lui     $a0, ($LC1 >> 16) # "Month: %d\n"
40 jalr    $t9
41 la      $a0, ($LC1 & 0xFFFF) # "Month: %d\n" ; 分岐遅延スロット
42 lw      $gp, 0x50+var_40($sp)
43 lw      $a1, 0x50+day($sp)
44 lw      $t9, (printf & 0xFFFF)($gp)
45 lui     $a0, ($LC2 >> 16) # "Day: %d\n"
46 jalr    $t9
47 la      $a0, ($LC2 & 0xFFFF) # "Day: %d\n" ; 分岐遅延スロット
48 lw      $gp, 0x50+var_40($sp)
49 lw      $a1, 0x50+hour($sp)
50 lw      $t9, (printf & 0xFFFF)($gp)
51 lui     $a0, ($LC3 >> 16) # "Hour: %d\n"
52 jalr    $t9
53 la      $a0, ($LC3 & 0xFFFF) # "Hour: %d\n" ; 分岐遅延スロット
54 lw      $gp, 0x50+var_40($sp)
55 lw      $a1, 0x50+minutes($sp)
56 lw      $t9, (printf & 0xFFFF)($gp)
57 lui     $a0, ($LC4 >> 16) # "Minutes: %d\n"
58 jalr    $t9
59 la      $a0, ($LC4 & 0xFFFF) # "Minutes: %d\n" ; 分岐遅延スロット
60 lw      $gp, 0x50+var_40($sp)
61 lw      $a1, 0x50+seconds($sp)
62 lw      $t9, (printf & 0xFFFF)($gp)
63 lui     $a0, ($LC5 >> 16) # "Seconds: %d\n"
64 jalr    $t9
65 la      $a0, ($LC5 & 0xFFFF) # "Seconds: %d\n" ; 分岐遅延スロット
66 lw      $ra, 0x50+var_4($sp)
67 or      $at, $zero ; ロード遅延スロット, NOP
68 jr      $ra
69 addiu   $sp, 0x50
70
71 $LC0:    .ascii "Year: %d\n"<0>
72 $LC1:    .ascii "Month: %d\n"<0>
73 $LC2:    .ascii "Day: %d\n"<0>
74 $LC3:    .ascii "Hour: %d\n"<0>
75 $LC4:    .ascii "Minutes: %d\n"<0>
76 $LC5:    .ascii "Seconds: %d\n"<0>

```

これは、分岐遅延スロットが私たちが混乱させる可能性のある例です。

たとえば、35行目に `addiu $a1, 1900` という命令があり、年に1900が加算されます。

これは34行目の対応する JALR の前に実行されますが、それを忘れないでください。

値の集合としての構造体

構造体が1つの場所に並んでいる変数であることを説明するために、*tm* 構造体の定義を再度見ながら、この例を再作してみましょう：リスト1.330

```
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday,
    ↵ , tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

注意：tm_sec フィールドへのポインタは、localtime_r、すなわち「構造体」の最初の要素に渡されます。

コンパイラは警告します。

Listing 1.334: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from ↵
    ↵ incompatible pointer type [enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but argument is of ↵
    ↵ type 'int *'
```

それにもかかわらず、これを生成します。

Listing 1.335: GCC 4.7.3

```
main      proc near
var_30     = dword ptr -30h
var_2C     = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec     = dword ptr -18h
tm_min     = dword ptr -14h
tm_hour    = dword ptr -10h
```

```

tm_mday = dword ptr -0Ch
tm_mon  = dword ptr -8
tm_year = dword ptr -4

    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    sub     esp, 30h
    call    __main
    mov     [esp+30h+var_30], 0 ; arg 0
    call    time
    mov     [esp+30h+unix_time], eax
    lea     eax, [esp+30h+tm_sec]
    mov     [esp+30h+var_2C], eax
    lea     eax, [esp+30h+unix_time]
    mov     [esp+30h+var_30], eax
    call    localtime_r
    mov     eax, [esp+30h+tm_year]
    add     eax, 1900
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_mon]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_mday]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_hour]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_min]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
    call    printf
    mov     eax, [esp+30h+tm_sec]
    mov     [esp+30h+var_2C], eax
    mov     [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
    call    printf
    leave
    retn
main      endp

```

このコードはこれまで見てきたものと同じもので、元のソースコードでは構造体であったか、単なる変数の集合であったかは言えません。

そしてこれは動きます。ただし、これを実際に行うことはお勧めしません。

通常、最適化されていないコンパイラは、関数内で宣言されたのと同じ順序で変数をローカルスタックに割り当てます。

とはいえ、何の保証也没有ありません。

ちなみに、tm_year、tm_mon、tm_mday、tm_hour、tm_min 変数については、他のコンパイラが警告することがありますが、tm_sec は初期化されずに使用されます。

実際、コンパイラは、これらが localtime_r() 関数で初期化されることを認識していません。

すべての構造体フィールドが int 型であるため、この例を選択しました。

SYSTEMTIME 構造体の場合のように、構造体フィールドが16ビット (WORD) の場合、これは機能しません。GetSystemTime() は、(ローカル変数が32ビット境界で整列されているため)、それらを正しく入力しません。次のセクションでそれについてもっと読む:「フィールドを構造体にパッキングする」(1.23.4 on page 437)

したがって、構造体は、1つの場所に並べて配置された単なる変数の集合です。構造体はコンパイラへの命令であり、変数を1つの場所に保持するように指示することができます。ところで、非常に初期のCバージョン (1972年以前) には、構造体がまったく存在しませんでした。[Dennis M. Ritchie, *The development of the C language*, (1993)]¹⁴⁹

デバッガの例はありません。すでに見たのとまったく同じです。

32ビットワードの配列としての構造体

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        int tmp=((int*)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    };
};
```

構造体へのポインタを int の配列にキャストします。そして、それは動作します! 例は2014年7月26日23:51:45に実行されます。

```
0x0000002D (45)
0x00000033 (51)
0x00000017 (23)
0x0000001A (26)
0x00000006 (6)
```

¹⁴⁹以下で利用可能 [pdf](#)

```

0x00000072 (114)
0x00000006 (6)
0x000000CE (206)
0x00000001 (1)

```

ここでの変数は、[1.330 on page 427](#) 構造体の定義で列挙されているのと同じ順序です。
コンパイル方法は次のとおりです。

Listing 1.336: 最適化 GCC 4.8.1

```

main proc near
    push    ebp
    mov     ebp, esp
    push    esi
    push    ebx
    and     esp, 0FFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; タイマー
    lea     ebx, [esp+14h]
    call    _time
    lea     esi, [esp+38h]
    mov     [esp+4], ebx      ; tp
    mov     [esp+10h], eax
    lea     eax, [esp+10h]
    mov     [esp], eax       ; タイマー
    call    _localtime_r
    nop
    lea     esi, [esi+0]      ; NOP
loc_80483D8:
; EBXは構造体へのポインタで、
; ESIは構造体の終わりへのポインタです。
    mov     eax, [ebx] ; get 配列から32ビットwordを取得
    add     ebx, 4      ; 構造体の次のフィールド
    mov     dword ptr [esp+4], offset a0x08xD ; "0x%08X (%d)\n"
    mov     dword ptr [esp], 1
    mov     [esp+0Ch], eax ; printf() に値を渡す
    mov     [esp+8], eax   ; printf() に値を渡す
    call    __printf_chk
    cmp     ebx, esi      ; 構造体の最後?
    jnz     short loc_80483D8 ; いいえ - 次の値をロードする
    lea     esp, [ebp-8]
    pop     ebx
    pop     esi
    pop     ebp
    retn
main endp

```

実際には、ローカルスタック内のスペースは最初に構造体として扱われ、その後で配列として扱われます。

このポインタを介して構造体のフィールドを変更することも可能です。

そして、この怪しいハッカーっぽいやり方は、プロダクトコードでの使用はお勧めしません。

練習問題

練習として、構造体を配列として扱い、現在の月の番号を変更（1つ増やしてください）してみてください。

配列オブジェクトとしての構造体

さらに進むことができます。ポインタをバイトの配列にキャストして、それをダンプしましょう。

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);

    localtime_r (&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf ("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf ("\n");
    }
};
```

```
0x2D 0x00 0x00 0x00
0x33 0x00 0x00 0x00
0x17 0x00 0x00 0x00
0x1A 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0x72 0x00 0x00 0x00
0x06 0x00 0x00 0x00
0xCE 0x00 0x00 0x00
0x01 0x00 0x00 0x00
```

この例は、2014年7月26日 23:51:45にも実行されています。¹⁵⁰ 値は以前のダンプと同じですが、(1.23.3 on page 434) もちろんこれはリトルエンディアンアーキテクチャなので、一番下のバイトが先になります。(?? on page ??)

¹⁵⁰ デモの目的で、日時は同じです。バイト値は固定されています。

Listing 1.337: 最適化 GCC 4.8.1

```

main proc near
    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
    and     esp, 0FFFFFF0h
    sub     esp, 40h
    mov     dword ptr [esp], 0 ; タイマー
    lea     esi, [esp+14h]
    call    _time
    lea     edi, [esp+38h] ; 構造体の終わり
    mov     [esp+4], esi ; tp
    mov     [esp+10h], eax
    lea     eax, [esp+10h]
    mov     [esp], eax ; タイマー
    call    _localtime_r
    lea     esi, [esi+0] ; NOP
; ESIはローカルスタックにある構造体へのポインタです。
; EDIは構造体の終わりへのポインタです。
loc_8048408:
    xor     ebx, ebx ; j=0

loc_804840A:
    movzx   eax, byte ptr [esi+ebx] ; バイトをロード
    add     ebx, 1 ; j=j+1
    mov     dword ptr [esp+4], offset a0x02x ; "0x%02X "
    mov     dword ptr [esp], 1
    mov     [esp+8], eax ; printf() にロードしたバイトを渡す
    call    __printf_chk
    cmp     ebx, 4
    jnz     short loc_804840A
; print carriage return character (CR)
    mov     dword ptr [esp], 0Ah ; c
    add     esi, 4
    call    _putchar
    cmp     esi, edi ; 構造体の終わり?
    jnz     short loc_8048408 ; j=0
    lea     esp, [ebp-0Ch]
    pop     ebx
    pop     esi
    pop     edi
    pop     ebp
    retn
main endp

```

第1.23.4節フィールドを構造体にパッキングする

1つ重要なことは、構造内のフィールドのパッキングです。

簡単な例を考えてみましょう：

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
    tmp.c=3;
    tmp.d=4;
    f(tmp);
};
```

見てきたように、2つの *char* フィールド（それぞれ1バイト）と2つの *int*（それぞれ4バイト）があります。

x86

このようにコンパイルされます。

Listing 1.338: MSVC 2012 /GS- /Ob0

```
1  _tmp$ = -16
2  _main PROC
3      push    ebp
4      mov     ebp, esp
5      sub     esp, 16
6      mov     BYTE PTR _tmp$[ebp], 1      ; フィールドaを設定
7      mov     DWORD PTR _tmp$[ebp+4], 2   ; フィールドbを設定
8      mov     BYTE PTR _tmp$[ebp+8], 3    ; フィールドcを設定
9      mov     DWORD PTR _tmp$[ebp+12], 4  ; フィールドdを設定
10     sub     esp, 16                    ; 一時的な構造体のために場所を確保
11     mov     eax, esp
12     mov     ecx, DWORD PTR _tmp$[ebp]   ; 構造体を一時的な場所にコピー
13     mov     DWORD PTR [eax], ecx
14     mov     edx, DWORD PTR _tmp$[ebp+4]
15     mov     DWORD PTR [eax+4], edx
16     mov     ecx, DWORD PTR _tmp$[ebp+8]
17     mov     DWORD PTR [eax+8], ecx
18     mov     edx, DWORD PTR _tmp$[ebp+12]
```

```

19     mov     DWORD PTR [eax+12], edx
20     call    _f
21     add     esp, 16
22     xor     eax, eax
23     mov     esp, ebp
24     pop     ebp
25     ret     0
26 _main     ENDP
27
28 _s$ = 8 ; size = 16
29 ?f@@YAXUs@@@Z PROC ; f
30     push    ebp
31     mov     ebp, esp
32     mov     eax, DWORD PTR _s$[ebp+12]
33     push    eax
34     movsx   ecx, BYTE PTR _s$[ebp+8]
35     push    ecx
36     mov     edx, DWORD PTR _s$[ebp+4]
37     push    edx
38     movsx   eax, BYTE PTR _s$[ebp]
39     push    eax
40     push    OFFSET $SG3842
41     call    _printf
42     add     esp, 20
43     pop     ebp
44     ret     0
45 ?f@@YAXUs@@@Z ENDP ; f
46 _TEXT     ENDS

```

構造全体を渡しますが、実際には、構造体は一時的な領域にコピーされて、(スタック内の領域は10行目に割り当てられ、次に4つのフィールドはすべて1つずつ、12行目から19行目にコピーされます) そのポインタ (アドレス) が渡されます。

f() 関数が構造体を変更するかどうか分からないため、構造体がコピーされます。それが変更された場合、main() の構造体はそのままではいけません。

私たちは C/C++ ポインタを使うことができました。結果のコードはほぼ同じですが、コピーは行いません。

次に見るように、各フィールドのアドレスは4バイトの境界に揃えられています。だからこそ、各 char が (int のように) 4バイトを占めるのです。なぜでしょうか？CPUが整列したアドレスでメモリにアクセスし、メモリからデータをキャッシュする方が簡単であるためです。

しかし、あまり経済的ではありません。

オプション (/Zp1) (nバイト境界で構造体をパックする /Zp[n]) でコンパイルしてみましょう。

Listing 1.339: MSVC 2012 /GS- /Zp1

```

1 _main     PROC
2     push    ebp
3     mov     ebp, esp
4     sub     esp, 12

```

```

5      mov     BYTE PTR _tmp$[ebp], 1      ; フィールドaを設定
6      mov     DWORD PTR _tmp$[ebp+1], 2    ; フィールドbを設定
7      mov     BYTE PTR _tmp$[ebp+5], 3     ; フィールドcを設定
8      mov     DWORD PTR _tmp$[ebp+6], 4    ; フィールドdを設定
9      sub     esp, 12                      ; 一時的な構造体のために場所を確保
10     mov     eax, esp
11     mov     ecx, DWORD PTR _tmp$[ebp]    ; 10バイトコピー
12     mov     DWORD PTR [eax], ecx
13     mov     edx, DWORD PTR _tmp$[ebp+4]
14     mov     DWORD PTR [eax+4], edx
15     mov     cx, WORD PTR _tmp$[ebp+8]
16     mov     WORD PTR [eax+8], cx
17     call    _f
18     add     esp, 12
19     xor     eax, eax
20     mov     esp, ebp
21     pop     ebp
22     ret     0
23 _main     ENDP
24
25 _TEXT     SEGMENT
26 _s$ = 8 ; size = 10
27 ?f@@YAXUs@@@Z PROC      ; f
28     push    ebp
29     mov     ebp, esp
30     mov     eax, DWORD PTR _s$[ebp+6]
31     push    eax
32     movsx   ecx, BYTE PTR _s$[ebp+5]
33     push    ecx
34     mov     edx, DWORD PTR _s$[ebp+1]
35     push    edx
36     movsx   eax, BYTE PTR _s$[ebp]
37     push    eax
38     push    OFFSET $SG3842
39     call    _printf
40     add     esp, 20
41     pop     ebp
42     ret     0
43 ?f@@YAXUs@@@Z ENDP      ; f

```

Now the structure takes only 10 bytes and each *char* value takes 1 byte. What does it give to us? Size economy. And as drawback —the CPU accessing these fields slower than it could.

構造体は10バイトしかなく、各 *char* 値は1バイト必要です。それは私たちに何を与えるのですか？サイズ経済。そして欠点として、CPUはこれらのフィールドにアクセスするのが遅くなります。

構造体も `main()` にコピーされます。フィールド単位ではなく、3つの `MOV` ペアを使用して直接10バイトをコピーします。なぜ4ではないのでしょうか？

コンパイラは、3つの `MOV` ペアを使用して10バイトをコピーする方が、2つの32ビットワードと4つの `MOV` ペアを使用して2バイトをコピーするよりも優れていると判断しました。

ちなみに、memcpy() 関数を呼び出す代わりに MOV を使用するようなコピーの実装は、memcpy() の呼び出しよりも速いため、広く使用されています。[?? on page ??](#)

簡単に推測できるように、構造体が多いソースファイルとオブジェクトファイルで使用されている場合、構造体パッキングについてはすべて同じ規則でコンパイルする必要があります。

各構造体フィールドの配置方法を設定するMSVC /Zp オプションの他に、#pragma pack コンパイラオプションもあります。このオプションはソースコード内で直接定義できます。MSVC¹⁵¹と GCC¹⁵² の両方で利用できます。

16ビットのフィールドで構成される SYSTEMTIME 構造体に戻りましょう。私たちのコンパイラは、1バイト境界でパックすることをどうやって知っていますか？

WinNT.h ファイルはこれを持っています：

Listing 1.340: WinNT.h

```
#include "pshpack1.h"
```

そしてこれを。

Listing 1.341: WinNT.h

```
#include "pshpack4.h" // 4バイトパッキングがデフォルト
```

PshPack1.h ファイルはこのようになっています。

Listing 1.342: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

コンパイラは #pragma pack の後で定義される構造体をパックする方法を知らせます。

¹⁵¹[MSDN: Working with Packing Structures](#)

¹⁵²[Structure-Packing Pragmas](#)

OllYDbg フィールドはデフォルトでパックされる

OillyDbg で我々の例を（フィールドがデフォルト（4バイト）で整列される）試してみましょう。

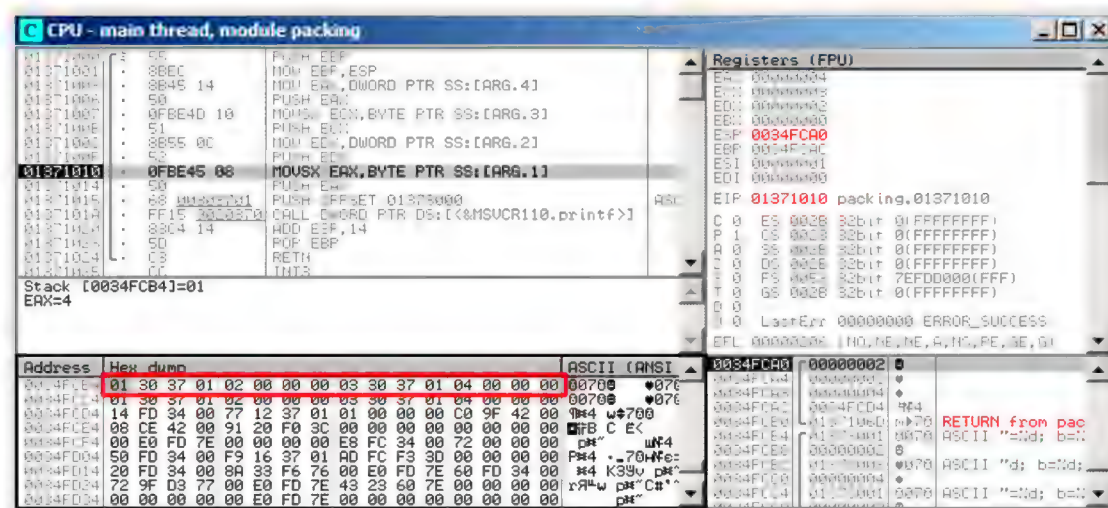


図 1.105: OllyDbg: printf() が実行される前

データウィンドウに4つフィールドが見えます。

しかし、ランダム値 (0x30, 0x37, 0x01) はどこから来たのでしょうか、最初の (a) と3番目のフィールド (c) の次でしょうか。

私たちの [1.338 on page 438](#) のリストを見ると、最初のフィールドと3番目のフィールドが `char` だと分かります。したがって、それぞれ1と3（6行目と8行目）が書かれています。

32ビットワードの残りの3バイトはメモリ内で変更されていません！したがって、ランダムなごみが残っています。

このゴミは printf() の出力には何の影響も与えません。その値は、MOVSX 命令を使用して準備されるため、ワードではなくバイトを使用します。リスト [1.338](#) (34行目と38行目)

ちなみに、`char` はMSVCとGCCでデフォルトでは符号ありなので、`MOVSX`（符号拡張）命令がここで使用されます。符号なしの`char` データ型または `uint8_t` がここで使用された場合、代わりに `MOVZX` 命令が使用されます。

OllyDbg 1バイト境界でアラインメントされるフィールド

ここでははっきりしています。4フィールドは10バイトを占め、値は並べて保存されます。

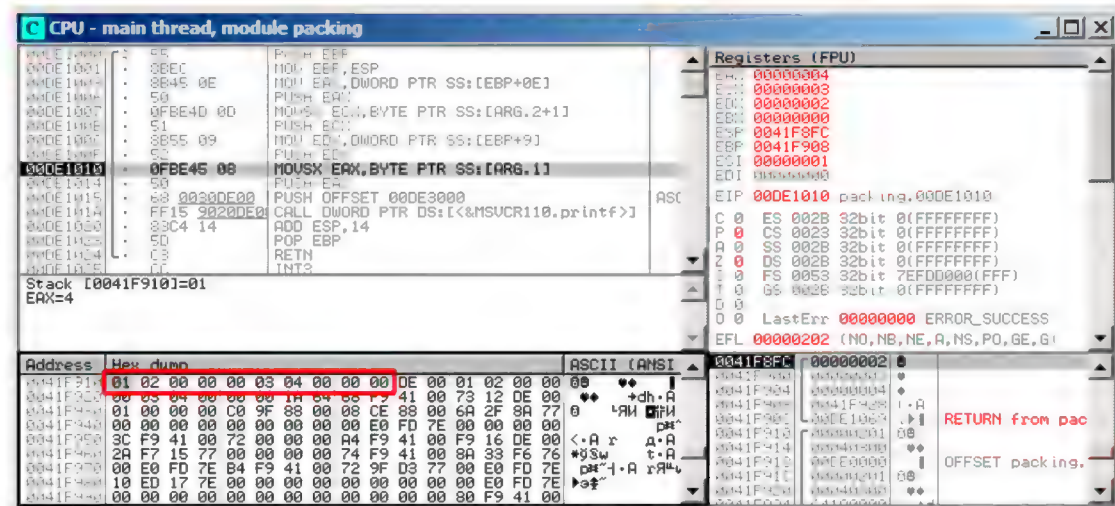


図 1.106: OllyDbg: printf() が実行される前

ARM

最適化 Keil 6/2013 (Thumbモード)

Listing 1.343: 最適化 Keil 6/2013 (Thumbモード)

```
.text:0000003E      exit ; CODE XREF: f+16
.text:0000003E 05 B0      ADD     SP, SP, #0x14
.text:00000040 00 BD      POP     {PC}

.text:00000280      f
.text:00000280
.text:00000280      var_18 = -0x18
.text:00000280      a      = -0x14
.text:00000280      b      = -0x10
.text:00000280      c      = -0xC
.text:00000280      d      = -8
.text:00000280
.text:00000280 0F B5      PUSH    {R0-R3,LR}
.text:00000282 81 B0      SUB     SP, SP, #4
.text:00000284 04 98      LDR     R0, [SP,#16] ; d
.text:00000286 02 9A      LDR     R2, [SP,#8] ; b
.text:00000288 00 90      STR     R0, [SP]
.text:0000028A 68 46      MOV     R0, SP
.text:0000028C 03 7B      LDRB    R3, [R0,#12] ; c
```

.text:0000028E 01 79	LDRB	R1, [R0,#4]	; a
.text:00000290 59 A0	ADR	R0, aADBDCDDD	; "a=%d; b=%d;
c=%d; d=%d\n"			
.text:00000292 05 F0 AD FF	BL	__2printf	
.text:00000296 D2 E6	B	exit	

思い出されるように、ここでは1つのポインタの代わりに構造体が渡されます。ARMの最初の4つの関数引数はレジスタを介して渡されるので、構造体のフィールドは R0-R3 を介して渡されます。

LDRB はメモリから1バイトをロードし、その符号を考慮して32ビットに拡張します。これはx86の MOVSX に似ています。ここでは、構造体からフィールド *a* および *c* をロードするために使用されます。

私たちが簡単に見つけたもう1つのことは、関数エピローグの代わりに、別の関数のエピローグにジャンプすることです！確かに、これはまったく異なる機能であり、私たちとは何ら関係がありませんでしたが、まったく同じエピローグを持っています（おそらく、ローカル変数を5つ含んでいるからです（ $5 \times 4 = 0x14$ ））。

また近くに位置しています（アドレスを見てください）。

実際、私たちが必要としているように動作すれば、どのエピローグが実行されるかは問題ではありません。

どうやら、Keilは別の関数の一部を再利用して節約するように決めているようです。

エピローグはジャンプに4バイト取ります。

ARM + 最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

Listing 1.344: 最適化 Xcode 4.6.3 (LLVM) (Thumb-2モード)

```
var_C = -0xC

    PUSH    {R7,LR}
    MOV     R7, SP
    SUB     SP, SP, #4
    MOV     R9, R1 ; b
    MOV     R1, R0 ; a
    MOVW    R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
    SXTB    R1, R1 ; prepare a
    MOVT.W  R0, #0
    STR     R3, [SP,#0xC+var_C] ; place d to stack for printf()
    ADD     R0, PC ; format-string
    SXTB    R3, R2 ; prepare c
    MOV     R2, R9 ; b
    BLX     _printf
    ADD     SP, SP, #4
    POP     {R7,PC}
```

SXTB (*Signed Extend Byte*) はx86の MOVSX に似ています。残りの部分はすべて同じです。

MIPS

Listing 1.345: 最適化 GCC 4.4.5 (IDA)

```

1 f:
2
3 var_18      = -0x18
4 var_10      = -0x10
5 var_4       = -4
6 arg_0       = 0
7 arg_4       = 4
8 arg_8       = 8
9 arg_C       = 0xC
10
11 ; $a0=s.a
12 ; $a1=s.b
13 ; $a2=s.c
14 ; $a3=s.d
15         lui    $gp, (__gnu_local_gp >> 16)
16         addiu   $sp, -0x28
17         la      $gp, (__gnu_local_gp & 0xFFFF)
18         sw      $ra, 0x28+var_4($sp)
19         sw      $gp, 0x28+var_10($sp)
20 ; 32ビットビッグエンディアン整数からバイトを準備
21         sra     $t0, $a0, 24
22         move    $v1, $a1
23 ; 32ビットビッグエンディアン整数からバイトを準備
24         sra     $v0, $a2, 24
25         lw      $t9, (printf & 0xFFFF)($gp)
26         sw      $a0, 0x28+arg_0($sp)
27         lui     $a0, ($LC0 >> 16) # "a=%d; b=%d; c=%d; d=%d\n"
28         sw      $a3, 0x28+var_18($sp)
29         sw      $a1, 0x28+arg_4($sp)
30         sw      $a2, 0x28+arg_8($sp)
31         sw      $a3, 0x28+arg_C($sp)
32         la      $a0, ($LC0 & 0xFFFF) # "a=%d; b=%d; c=%d; d=%d\n"
33         move    $a1, $t0
34         move    $a2, $v1
35         jalr    $t9
36         move    $a3, $v0 ; 分岐遅延スロット
37         lw      $ra, 0x28+var_4($sp)
38         or      $at, $zero ; ロード遅延スロット, NOP
39         jr      $ra
40         addiu   $sp, 0x28 ; 分岐遅延スロット
41
42 $LC0:      .ascii "a=%d; b=%d; c=%d; d=%d\n"<0>

```

構造体フィールドはレジスタ \$A0..\$A3 に入ってから printf() のために \$A1..\$A3 に再整理され、4番目のフィールド (\$A3 から) は SW を使ってローカルスタックを経由して渡されます。

しかし、2つのSRA (「Shift Word Right Arithmetic」) 命令があり、これは char フィールドを準備します。なぜでしょうか？

MIPSはデフォルトではビッグエンディアンアーキテクチャです?? on page ??。私たちが動かすDebian Linuxもビッグエンディアンです。

したがって、バイト変数が32ビット構造のスロットに格納されるとき、それらは高位31～24ビットを占有します。

また、*char* 変数を32ビット値に拡張する必要がある場合は、それを24ビット右にシフトする必要があります。

char は符号付き型なので、ここでは論理シフトの代わりに算術シフトが使用されます。

もう一言

関数の引数として構造体を渡すのは（構造体へのポインタを渡すのではなく）構造体のフィールドを 1つ1つ渡すのと同じです。

構造体のフィールドがデフォルトでパックされる場合、*f()* 関数は以下のように書き換える可能です。

```
void f(char a, int b, char c, int d)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
};
```

そして同じコードになります。

第1.23.5節構造体の入れ子

さて、ある構造体が別の構造体の内部で定義される状況はどうでしょうか

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
           s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};

int main()
```

```
{
    struct outer_struct s;
    s.a=1;
    s.b=2;
    s.c.a=100;
    s.c.b=101;
    s.d=3;
    s.e=4;
    f(s);
};
```

...この場合、inner_struct フィールドは両方とも outer_struct のフィールドa,bとd,eの間に位置します。

コンパイルしてみましょう (MSVC 2010)。

Listing 1.346: 最適化 MSVC 2010 /Ob0

```
$SG2802 DB      'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT      SEGMENT
_s$ = 8
_f        PROC
    mov     eax, DWORD PTR _s$[esp+16]
    movsx   ecx, BYTE PTR _s$[esp+12]
    mov     edx, DWORD PTR _s$[esp+8]
    push    eax
    mov     eax, DWORD PTR _s$[esp+8]
    push    ecx
    mov     ecx, DWORD PTR _s$[esp+8]
    push    edx
    movsx   edx, BYTE PTR _s$[esp+8]
    push    eax
    push    ecx
    push    edx
    push    OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call    _printf
    add     esp, 28
    ret     0
_f        ENDP

_s$ = -24
_main     PROC
    sub     esp, 24
    push    ebx
    push    esi
    push    edi
    mov     ecx, 2
    sub     esp, 24
    mov     eax, esp
; この瞬間から、EAXはESPと同義
    mov     BYTE PTR _s$[esp+60], 1
    mov     ebx, DWORD PTR _s$[esp+60]
    mov     DWORD PTR [eax], ebx
```

```

mov     DWORD PTR [eax+4], ecx
lea     edx, DWORD PTR [ecx+98]
lea     esi, DWORD PTR [ecx+99]
lea     edi, DWORD PTR [ecx+2]
mov     DWORD PTR [eax+8], edx
mov     BYTE PTR _s$[esp+76], 3
mov     ecx, DWORD PTR _s$[esp+76]
mov     DWORD PTR [eax+12], esi
mov     DWORD PTR [eax+16], ecx
mov     DWORD PTR [eax+20], edi
call    _f
add     esp, 24
pop     edi
pop     esi
xor     eax, eax
pop     ebx
add     esp, 24
ret     0
_main   ENDP

```

ここで1つ興味深いのは、このアセンブリコードを調べると、内部に別の構造体が使用されていることさえわからないということです。従って、入れ子の構造体は一次のまたは一次元の構造体に展開されていると言えます。

もちろん、`struct inner_struct c;` 宣言を `struct inner_struct *c;` で置き換えるとしたら、(従って、ここでポインタを作成する) 状況はまったく違ってきます。

OllyDbg

例を OllyDbg にロードしてメモリ上の `outer_struct` を見てみましょう。

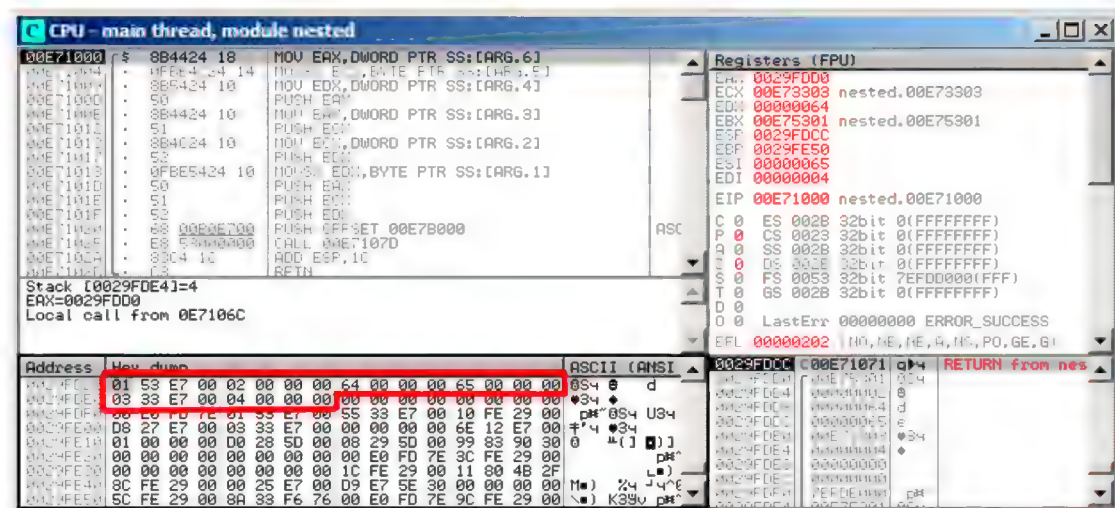


図 1.107: OllyDbg: `printf()` が実行される前

値がメモリ上にどのようにあるかを示しています。

- `(outer_struct.a)` (byte) 1 + 3バイトのランダムなゴミの値
- `(outer_struct.b)` (32-bitワード) 2;
- `(inner_struct.a)` (32-bitワード) 0x64 (100);
- `(inner_struct.b)` (32-bitワード) 0x65 (101);
- `(outer_struct.d)` (byte) 3 + 3バイトのランダムなゴミの値
- `(outer_struct.e)` (32-bit word) 4.

第1.23.6節構造体のビットフィールド

CPUIDの例

C/C++ 言語では、各構造体フィールドの正確なビット数を定義できます。メモリ空間を節約する必要がある場合に非常に便利です。たとえば、`bool` 変数には1ビットで十分です。しかし、スピードが重要なら合理的ではありません。

CPUID¹⁵³命令の例を考えてみましょう。この命令は、現在のCPUとその機能に関する情報を返します。

命令が実行される前に EAX が1に設定されている場合、CPUID は EAX レジスタに情報がパックされて返ります。

¹⁵³[wikipedia](#)

3:0 (4 bits)	ステッピング
7:4 (4 bits)	モデル
11:8 (4 bits)	ファミリー
13:12 (2 bits)	プロセッサタイプ
19:16 (4 bits)	拡張モデル
27:20 (8 bits)	拡張ファミリー

MSVC 2010には CPUID マクロがありますが、GCC 4.4.1にはありません。ですから組み込みアセンブラ¹⁵⁴の助けを借りてGCCのためにこの機能を自分自身で作ってみましょう。

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
    asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
```

¹⁵⁴[GCCアセンブラ内部の詳細](#)

```

printf ("processor_type=%d\n", tmp->processor_type);
printf ("extended_model_id=%d\n", tmp->extended_model_id);
printf ("extended_family_id=%d\n", tmp->extended_family_id);

return 0;
};

```

CPUID が EAX/EBX/ECX/EDX を満たすと、これらのレジスタは `b[]` 配列に書き込まれます。次に、`CPUID_1_EAX` 構造体へのポインタを持ち、それを `b[]` 配列から EAX の値に向けます。

つまり、32ビットの *int* 値を構造体として扱います。次に、構造体から特定のビットを読み込みます。

MSVC

/Ox オプションを付けてMSVC 2008でコンパイルしてみましょう。

Listing 1.347: 最適化 MSVC 2008

```

_b$ = -16 ; size = 16
_main PROC
    sub     esp, 16
    push    ebx

    xor     ecx, ecx
    mov     eax, 1
    cpushid
    push    esi
    lea     esi, DWORD PTR _b$[esp+24]
    mov     DWORD PTR [esi], eax
    mov     DWORD PTR [esi+4], ebx
    mov     DWORD PTR [esi+8], ecx
    mov     DWORD PTR [esi+12], edx

    mov     esi, DWORD PTR _b$[esp+24]
    mov     eax, esi
    and     eax, 15
    push    eax
    push    OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call    _printf

    mov     ecx, esi
    shr     ecx, 4
    and     ecx, 15
    push    ecx
    push    OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call    _printf

    mov     edx, esi
    shr     edx, 8
    and     edx, 15

```

```

push    edx
push    OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
call    _printf

mov     eax, esi
shr     eax, 12
and     eax, 3
push    eax
push    OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
call    _printf

mov     ecx, esi
shr     ecx, 16
and     ecx, 15
push    ecx
push    OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
call    _printf

shr     esi, 20
and     esi, 255
push    esi
push    OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
call    _printf
add     esp, 48
pop     esi

xor     eax, eax
pop     ebx

add     esp, 16
ret     0
_main   ENDP

```

SHR 命令は、EAX 内の値を、スキップしなければならないビット数だけシフトします。例えば、右側のビットを無視します。

AND 命令は、左側の不要ビットをクリアします。言い換えれば、必要な EAX レジスタのビットだけを残します。

MSVC + OllyDbg

OllyDbg にサンプルをロードして、CUIDの実行後にEAX/EBX/ECX/EDXにどのような値が設定されているかを見てみましょう。

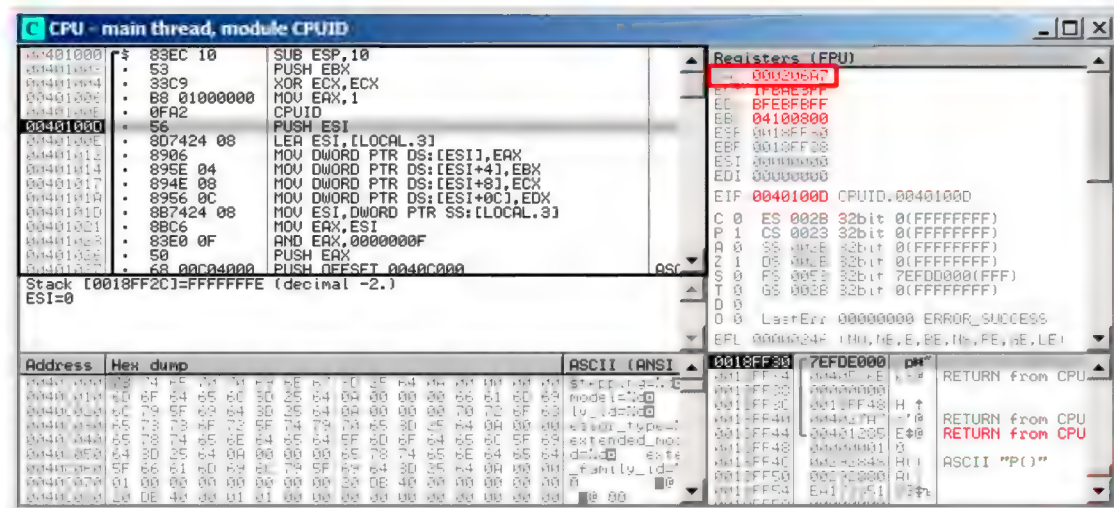


図 1.108: OllyDbg: CUID実行後

EAXは 0x000206A7(CPUはIntel Xeon E3-1220)です。二進数では 0b0000000000000001000000110101001111です。

ビットがフィールドにどのように分配されるかは次のとおりです。

フィールド	二進数	十進数
reserved2	0000	0
extended_family_id	00000000	0
extended_model_id	0010	2
reserved1	00	0
processor_id	00	0
family_id	0110	6
model	1010	10
stepping	0111	7

Listing 1.348: Console output

```
stepping=7
model=10
family_id=6
processor_type=0
extended_model_id=2
extended_family_id=0
```

GCC

-03 オプション付きのGCC 4.4.1を試してみましょう。

Listing 1.349: 最適化 GCC 4.4.1

```

main          proc near ; DATA XREF: _start+17
    push      ebp
    mov       ebp, esp
    and       esp, 0FFFFFFF0h
    push      esi
    mov       esi, 1
    push      ebx
    mov       eax, esi
    sub       esp, 18h
    cpuid
    mov       esi, eax
    and       eax, 0Fh
    mov       [esp+8], eax
    mov       dword ptr [esp+4], offset aSteppingD ; "stepping=%d\n"
    mov       dword ptr [esp], 1
    call      __printf_chk
    mov       eax, esi
    shr       eax, 4
    and       eax, 0Fh
    mov       [esp+8], eax
    mov       dword ptr [esp+4], offset aModelD ; "model=%d\n"
    mov       dword ptr [esp], 1
    call      __printf_chk
    mov       eax, esi
    shr       eax, 8
    and       eax, 0Fh
    mov       [esp+8], eax
    mov       dword ptr [esp+4], offset aFamily_idD ; "family_id=%d\n"
    mov       dword ptr [esp], 1
    call      __printf_chk
    mov       eax, esi
    shr       eax, 0Ch
    and       eax, 3
    mov       [esp+8], eax
    mov       dword ptr [esp+4], offset aProcessor_type ; "processor_type=%d\n"
    mov       dword ptr [esp], 1
    call      __printf_chk
    mov       eax, esi
    shr       eax, 10h
    shr       esi, 14h
    and       eax, 0Fh
    and       esi, 0FFh
    mov       [esp+8], eax
    mov       dword ptr [esp+4], offset aExtended_model ;
    "extended_model_id=%d\n"
    mov       dword ptr [esp], 1
    call      __printf_chk
    mov       [esp+8], esi

```

```

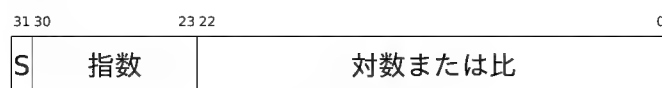
mov     dword ptr [esp+4], offset unk_80486D0
mov     dword ptr [esp], 1
call    __printf_chk
add     esp, 18h
xor     eax, eax
pop     ebx
pop     esi
mov     esp, ebp
pop     ebp
retn
main                                         endp

```

ほとんど同じです。唯一注目すべきは、GCCは、`printf()` の各呼び出しの前に個別に計算するのではなく、`extended_model_id` と `extended_family_id` の計算をどういうわけか1つのブロックに組み合わせることでです。

フロート型のデータを構造体として扱う

FPUについてのセクション ([1.19 on page 268](#)) ですでに述べたように、*float* と *double* の両方は、符号、仮数部（または小数部）、指数部で構成されます。しかし、これらのフィールドを直接編集することはできるでしょうか？*float* で試してみましょう。



(S — 記号)

```

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // 端数部分
    unsigned int exponent : 8;  // 指数 + 0x3FF
    unsigned int sign : 1;      // 符号ビット
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // 負号を設定
    t.exponent=t.exponent+2; // dに 2^n(nはここでは2) を乗算

    memcpy (&f, &t, sizeof (float));
}

```

```

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};

```

`float_as_struct` 構造体は、`float` と同じ量のメモリ、すなわち4バイトまたは32ビットを占有します。

ここで、入力値に負の符号を設定し、指数に2を加えることによって、整数を 2^2 、すなわち4で乗算します。

最適化をオンにしないでMSVC 2008でコンパイルしましょう：

Listing 1.350: 非最適化 MSVC 2008

```

_t$ = -8    ; size = 4
_f$ = -4    ; size = 4
_in$ = 8    ; size = 4
?f@@YAMM@Z PROC ; f
    push    ebp
    mov     ebp, esp
    sub     esp, 8

    fld     DWORD PTR __in$[ebp]
    fstp    DWORD PTR _f$[ebp]

    push    4
    lea     eax, DWORD PTR _f$[ebp]
    push    eax
    lea     ecx, DWORD PTR _t$[ebp]
    push    ecx
    call    _memcpy
    add     esp, 12

    mov     edx, DWORD PTR _t$[ebp]
    or      edx, -2147483648 ; 80000000H - マイナスの符号を設定
    mov     DWORD PTR _t$[ebp], edx

    mov     eax, DWORD PTR _t$[ebp]
    shr     eax, 23          ; 00000017H - 指数を落とす
    and     eax, 255         ; 000000ffH - 指数のみを残す
    add     eax, 2           ; 2を加算
    and     eax, 255         ; 000000ffH
    shl     eax, 23          ; 00000017H - ビット30:23の場所に結果をシフトする
    mov     ecx, DWORD PTR _t$[ebp]
    and     ecx, -2139095041 ; 807ffffFH - 指数を落とす

; 新しく計算された指数で指数なしの元の値を追加する
    or      ecx, eax
    mov     DWORD PTR _t$[ebp], ecx

```

```

push    4
lea     edx, DWORD PTR _t$[ebp]
push    edx
lea     eax, DWORD PTR _f$[ebp]
push    eax
call    _memcpy
add     esp, 12

fld     DWORD PTR _f$[ebp]

mov     esp, ebp
pop     ebp
ret     0
?f@@YAMM@Z ENDP    ; f

```

少し冗長です。/Ox フラグを付けてコンパイルした場合、memcpy() 呼び出しはなく、f 変数が直接使用されます。しかし、最適化されていないバージョンを見れば分かりやすくなります。

GCC 4.4.1を -O3 つきで実行したらどうなりますか？

Listing 1.351: 最適化 GCC 4.4.1

```

; f(float)
public _Z1ff
_Z1ff proc near

var_4 = dword ptr -4
arg_0 = dword ptr 8

    push    ebp
    mov     ebp, esp
    sub     esp, 4
    mov     eax, [ebp+arg_0]
    or      eax, 80000000h    ; マイナス符号を設定
    mov     edx, eax
    and     eax, 807FFFFFFh   ; EAXに符号と仮数部のみを残す
    shr     edx, 23           ; 指数を準備
    add     edx, 2            ; 2を加算
    movzx   edx, dl           ; EDXの7:0を除くビットをすべてクリア
    shl     edx, 23          ; 新しく計算された指数をその場所に移す
    or      eax, edx         ; 指数なしで新しい指数と元の値を結合する
    mov     [ebp+var_4], eax
    fld     [ebp+var_4]
    leave
    retn
_Z1ff endp

public main
main proc near
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h

```

```

    sub    esp, 10h
    fld    ds:dword_8048614 ; -4.936
    fstp   qword ptr [esp+8]
    mov    dword ptr [esp+4], offset asc_8048610 ; "%f\n"
    mov    dword ptr [esp], 1
    call   __printf_chk
    xor    eax, eax
    leave
    retn
main     endp

```

f() 関数はほぼ理解できます。しかし、興味深いのは、GCCが構造体フィールドを持つこのような問題にかかわらず、コンパイル時に f(1.234) の結果を計算でき、コンパイル時に事前計算されて printf() にこの引数を用意したことです。

第1.23.7節練習問題

- <http://challenges.re/71>
- <http://challenges.re/72>

第1.24節共用体

C/C++ 共用体は、あるデータ型の変数（またはメモリブロック）を別のデータ型の変数として解釈するために使用されます。

第1.24.1節擬似乱数生成器の例

0と1の間の浮動小数点の乱数が必要な場合、最も簡単なのはメルセンヌツイスターのような [PRNG](#)¹⁵⁵ を使うことです。ランダムな符号なし32ビット値を生成します（つまり、ランダム32ビットを生成します）。この値をfloatに変換し、RAND_MAX（ここでは 0xFFFFFFFF）で割ります。我々は0..1の間で値を取得します。

しかし知ってのとおり、除算は遅いです。また、できるだけ少ないFPU演算で実行したいと考えています。私たちは除算を取り除くことができるでしょうか？

浮動小数点数が符号ビット、仮数ビット、指数ビットからなるものを思い出してみましょう。ランダムな浮動小数点数を得るには、すべての仮数ビットにランダムなビットを格納するだけです。

指数部はゼロではありません（浮動小数点はこの場合非正規化されています）ので、指数部に0b01111111 を格納しています。指数部が1であることを意味します。次に、仮数部をランダムビットで埋め、符号ビットを0に設定する（正の数）と出来上がり。生成される数は1と2の間にあるので、1を減算する必要があります。

私の例では、非常に単純な線形合同乱数ジェネレータが使用され、¹⁵⁶ これは32ビットの数値を生成します。PRNGはUNIXのタイムスタンプ形式で現在の時刻で初期化されます。

¹⁵⁵ 擬似乱数生成器

¹⁵⁶ アイデアは以下から取りました: [URL](#)

ここでは *float* 型を *union* として表します。これは、メモリの種類を異なる型として解釈できる C/C++ 構造です。私たちの場合、*union* 型の変数を作成し、それを *float* または *uint32_t* のようにアクセスすることができます。それはまさに汚いハックだと言えるでしょう。

整数PRNGコードは、すでに検討しているものと同じです：[1.22 on page 411](#) このコードはコンパイルされた形式では省略されています。

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// 整数PRNG定義、データとルーチン

// ニューメリカルレシピ本からとった定数
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // グローバル変数

void my_srand(uint32_t i)
{
    RNG_state=i;
};

uint32_t my_rand()
{
    RNG_state=RNG_state*RNG_a+RNG_c;
    return RNG_state;
};

// FPU PRNG 定義とルーチン

union uint32_t_float
{
    uint32_t i;
    float f;
};

float float_rand()
{
    union uint32_t_float tmp;
    tmp.i=my_rand() & 0x007fffff | 0x3f800000;
    return tmp.f-1;
};

// テスト

int main()
{
    my_srand(time(NULL)); // PRNG 初期化

    for (int i=0; i<100; i++)
        printf ("%f\n", float_rand());
}
```

```

    return 0;
};

```

x86

Listing 1.352: 最適化 MSVC 2010

```

$SG4238 DB      '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff000000000000r    ; 1

tv130 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
    push    ecx
    call    ?my_rand@@YAIXZ
; EAX=疑似乱数値
    and     eax, 8388607                ; 007ffffFH
    or      eax, 1065353216            ; 3f800000H
; EAX=疑似乱数値 & 0x007fffff | 0x3f800000
; ローカルスタックに保存
    mov     DWORD PTR _tmp$(esp+4), eax
; 浮動小数点数としてリロード
    fld     DWORD PTR _tmp$(esp+4)
; 1.0を引く
    fsub    QWORD PTR __real@3ff0000000000000
; ローカルスタックに格納された値を格納してリロードする
; これらの命令は冗長:
    fstp    DWORD PTR tv130(esp+4)
    fld     DWORD PTR tv130(esp+4)
    pop     ecx
    ret     0
?float_rand@@YAMXZ ENDP

_main PROC
    push    esi
    xor     eax, eax
    call    _time
    push    eax
    call    ?my_srand@@YAXI@Z
    add     esp, 4
    mov     esi, 100
$LL3@main:
    call    ?float_rand@@YAMXZ
    sub     esp, 8
    fstp    QWORD PTR [esp]
    push    OFFSET $SG4238
    call    _printf
    add     esp, 12
    dec     esi
    jne     SHORT $LL3@main
    xor     eax, eax
    pop     esi

```



```

    ret    0
_main    ENDP

```

この例はC++としてコンパイルされており、これはC++での名前の変換であるため、ここでは関数名が非常に奇妙です。これについては後で説明します：[?? on page ??](#) これをMSVC 2012でコンパイルすると、FPU用のSIMD命令が使用されます。詳細については、[こちらを参照してください：1.29.5 on page 535](#)

ARM (ARMモード)

Listing 1.353: 最適化 GCC 4.6.3 (IDA)

```

float_rand
    STMFD    SP!, {R3,LR}
    BL      my_rand
; R0=疑似乱数値
    FLDS     S0, =1.0
; S0=1.0
    BIC      R3, R0, #0xFF000000
    BIC      R3, R3, #0x800000
    ORR      R3, R3, #0x3F800000
; R3=疑似乱数値 & 0x007fffff | 0x3f800000
; R3からFPU (レジスタS15) にコピー
; それはビット単位のコピーのように動作し、変換は行われません
    FMSR     S15, R3
; 1.0を引いて結果をS0に残す
    FSUBS    S0, S15, S0
    LDMFD    SP!, {R3,PC}

flt_5C      DCFS 1.0

main
    STMFD    SP!, {R4,LR}
    MOV      R0, #0
    BL      time
    BL      my_srand
    MOV      R4, #0x64 ; 'd'

loc_78
    BL      float_rand
; S0=疑似乱数値
    LDR      R0, =aF ; "%f"
; float型の値をdouble型の値に変換する (printf() が必要とする)
    FCVTDS   D7, S0
; D7からレジスタR2/R3ペアへのビット単位のコピー (printf() 用)
    FMRRD    R2, R3, D7
    BL      printf
    SUBS     R4, R4, #1
    BNE     loc_78
    MOV      R0, R4
    LDMFD    SP!, {R4,PC}

aF          DCB "%f",0xA,0

```

また、objdumpにダンプを作成し、FPU命令の名前が **IDA** とは異なることを確認します。見たところ、IDAとbinutilsの開発者は異なるマニュアルを使ったのでしょうか？おそらく、両方の命令の変種を知っておくとよいでしょう。

Listing 1.354: 最適化 GCC 4.6.3 (objdump)

```
00000038 <float_rand>:
 38: e92d4008      push    {r3, lr}
 3c: ebfffffe      bl      10 <my_rand>
 40: ed9f0a05      vldr    s0, [pc, #20] ; 5c <float_rand+0x24>
 44: e3c034ff      bic     r3, r0, #-16777216 ; 0xff000000
 48: e3c33502      bic     r3, r3, #8388608 ; 0x800000
 4c: e38335fe      orr     r3, r3, #1065353216 ; 0x3f800000
 50: ee073a90      vmov    s15, r3
 54: ee370ac0      vsub.f32 s0, s15, s0
 58: e8bd8008      pop     {r3, pc}
 5c: 3f800000      svccc   0x00800000

00000000 <main>:
 0: e92d4010      push    {r4, lr}
 4: e3a00000      mov     r0, #0
 8: ebfffffe      bl      0 <time>
 c: ebfffffe      bl      0 <main>
10: e3a04064      mov     r4, #100 ; 0x64
14: ebfffffe      bl      38 <main+0x38>
18: e59f0018      ldr     r0, [pc, #24] ; 38 <main+0x38>
1c: eeb77ac0      vcvtf64.f32 d7, s0
20: ec532b17      vmov    r2, r3, d7
24: ebfffffe      bl      0 <printf>
28: e2544001      subs    r4, r4, #1
2c: 1afffff8      bne     14 <main+0x14>
30: e1a00004      mov     r0, r4
34: e8bd8010      pop     {r4, pc}
38: 00000000      andeq   r0, r0, r0
```

float_rand() の0x5cと main() の0x38の命令は、(疑似) 乱数ノイズです。

第1.24.2節 計算機イプシロンを計算する

計算機イプシロンは、**FPU**が使用できる最小の値です。浮動小数点数に割り当てられるビットが多いほど、計算機イプシロンは小さくなります。*float* では $2^{-23} \approx 1.19e-07$ で、*double* では $2^{-52} \approx 2.22e-16$ です。[Wikipediaの記事](#) も参照してください。

興味深いことに、計算機イプシロンを計算するのはとても簡単です。

```
#include <stdio.h>
#include <stdint.h>

union uint_float
{
    uint32_t i;
```

```

        float f;
    };

float calculate_machine_epsilon(float start)
{
    union uint_float v;
    v.f=start;
    v.i++;
    return v.f-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
};

```

ここで行うことは、IEEE 754形式の数の小数部分を整数として扱い、それに1を加えることです。結果の浮動小数点数は *starting_value + machine_epsilon* に等しいので、測定するために（浮動小数点演算を使用して）開始値を減算する必要があります。1ビットが単精度（*float*）にどのように反映されるかを測定します。共用体は、ここでは通常の整数としてIEEE 754形式の数にアクセスする方法として機能します。1を加えることは実際には数の小数部分に1を加えますが、言うまでもなく、オーバーフローは可能であり、指数部分に1を加えることになります。

x86

Listing 1.355: 最適化 MSVC 2010

```

tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
    fld     DWORD PTR _start$[esp-4]
; この命令は冗長:
    fst     DWORD PTR _v$[esp-4]
    inc     DWORD PTR _v$[esp-4]
    fsubr   DWORD PTR _v$[esp-4]
; この命令ペアも冗長:
    fstp    DWORD PTR tv130[esp-4]
    fld     DWORD PTR tv130[esp-4]
    ret     0
_calculate_machine_epsilon ENDP

```

2番目の FST 命令は冗長です。入力値を同じ場所に格納する必要はありません（コンパイラは、ローカルスタックの入力引数と同じ位置に *v* 変数を割り当てることにしました）。それは通常の整数の変数なので、INC でインクリメントされます。その後、32ビットのIEEE 754形式の数としてFPUにロードされ、FSUBR が残りのジョブを実行し、結果の値が ST0 に格納されます。最後の FSTP/FLD 命令ペアは冗長ですが、コンパイラは最適化しませんでした。

ARM64

例を64ビットに拡張してみましょう。

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
    uint64_t i;
    double d;
} uint_double;

double calculate_machine_epsilon(double start)
{
    uint_double v;
    v.d=start;
    v.i++;
    return v.d-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
};
```

ARM64にはFPUのDレジスタに数値を加算する命令がないため、入力値（D0 に入力されたもの）が最初にGPRにコピーされ、インクリメントされ、FPUレジスタの D1 にコピーされてから減算が行われます。

Listing 1.356: 最適化 GCC 4.9 ARM64

```
calculate_machine_epsilon:
    fmov    x0, d0          ; double型の入力値をX0にロード
    add     x0, x0, 1       ; X0++
    fmov    d1, x0          ; FPUレジスタにムーブ
    fsub    d0, d1, d0     ; 引く
    ret
```

SIMD命令を使用してx64用にコンパイルされたこの例も参照してください：[1.29.4 on page 534](#)

MIPS

ここでの新しい命令は MTC1 （「Move To Coprocessor 1」）です。GPRからFPUのレジスタへデータを転送するだけです。

Listing 1.357: 最適化 GCC 4.4.5 (IDA)

```
calculate_machine_epsilon:
    mfc1    $v0, $f12
    or      $at, $zero ; NOP
    addiu   $v1, $v0, 1
    mtc1    $v1, $f2
```

```

jr      $ra
sub.s   $f0, $f2, $f12 ; branch delay slot

```

結論

誰かがこのトリックを実際のコードで必要とするかどうかはわかりにくいですが、この本で何度も述べたように、この例は IEEE 754形式と C/C++ の共用体を説明するのに役立ちます。

第1.24.3節FSCALE replacement

Agner Fog氏による *Optimizing subroutines in assembly language / An optimization guide for x86 platforms* では¹⁵⁷、多くのCPUでは FSCALE FPU命令 (2^n の計算) が遅くなる可能性があるとして述べ、より速いものを提案しています。

これが私のアセンブリコードの C/C++ への翻訳です。

```

#include <stdint.h>
#include <stdio.h>

union uint_float
{
    uint32_t i;
    float f;
};

float flt_2n(int N)
{
    union uint_float tmp;

    tmp.i=(N<<23)+0x3f800000;
    return tmp.f;
};

struct float_as_struct
{
    unsigned int fraction : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
};

float flt_2n_v2(int N)
{
    struct float_as_struct tmp;

    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x7f;
    return *(float*)&tmp;
};

```

¹⁵⁷http://www.agner.org/optimize/optimizing_assembly.pdf

```

union uint64_double
{
    uint64_t i;
    double d;
};

double dbl_2n(int N)
{
    union uint64_double tmp;

    tmp.i=((uint64_t)N<<52)+0x3ff0000000000000UL;
    return tmp.d;
};

struct double_as_struct
{
    uint64_t fraction : 52;
    int exponent : 11;
    int sign : 1;
};

double dbl_2n_v2(int N)
{
    struct double_as_struct tmp;

    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x3ff;
    return *(double*)&tmp;
};

int main()
{
    // 211 = 2048
    printf ("%f\n", flt_2n(11));
    printf ("%f\n", flt_2n_v2(11));
    printf ("%lf\n", dbl_2n(11));
    printf ("%lf\n", dbl_2n_v2(11));
};

```

FSCALE 命令はあなたの環境ではより速いかもしれませんが、それでも、それは共用体の良い例であり、指数が 2^n 形式で格納されるという事実です。そのため、入力された n の値はIEEE 754形式で符号化された数の指数にシフトされます。その後、0x3f800000または0x3ff0000000000000を追加して指数を補正します。

構造体を使用してシフトなしで同じことを実行できますが、それでも内部ではシフト操作が発生しました。

第1.24.4節高速平方根計算

float が整数として解釈される別のよく知られたアルゴリズムは平方根の高速計算です。

Listing 1.358: ソースコードはウィキペディアから取りました: https://en.wikipedia.org/wiki/Methods_of_computing_square_root

```
/* floatはIEEE 754形式の単精度浮動小数点数で、
 * intは32ビットです。 */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* 同じビットだが、intとして扱われる */
    /*
     * 次のコードを正当化するため、以下を証明せよ
     *
     * (((val_int / 2^m) - b) / 2) + b) * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m
     *
     * 以下の条件において
     *
     * b = 指数バイアス
     * m = 仮数ビットの数
     *
     * .
     */

    val_int -= 1 << 23; /* 2^m を除算 */
    val_int >>= 1; /* 2で除算 */
    val_int += 1 << 29; /* ((b + 1) / 2) * 2^m を加算 */

    return *(float*)&val_int; /* float型として再び解釈 */
}
```

演習として、この関数をコンパイルして、その機能を理解することを試みることができます。

$\frac{1}{\sqrt{x}}$ の高速計算のよく知られたアルゴリズムもあります。Quake III Arenaで使用されていたため、アルゴリズムが普及したと考えられます。

アルゴリズムの説明はWikipediaにあります: http://en.wikipedia.org/wiki/Fast_inverse_square_root

第1.25節関数へのポインタ

関数へのポインタは、他のポインタと同様に、コードセグメント内の関数の開始アドレスにすぎません。

それらはコールバック関数を呼び出すためによく使われます。

よく知られている例は次のとおりです。

- 標準Cライブラリの `qsort()`, `atexit()`
- *NIX OS シグナル
- スレッド開始: `CreateThread()` (win32), `pthread_create()` (POSIX);
- 多くのwin32関数、例えば `EnumChildWindows()`.

- Linuxカーネル内部の色々なところで、例えばファイルシステムドライバ関数はコールバック経由で呼び出されます。
- GCCプラグイン関数もコールバック経由で呼び出されます。

そのため、`qsort()` 関数は C/C++ 標準ライブラリのquicksortの実装です。これら2つの要素を比較する関数があり、`qsort()` が関数を呼び出すことができる限り、関数はあらゆるデータ、あらゆるタイプのデータをソートすることができます。

比較関数は次のように定義できます。

```
int (*compare)(const void *, const void *)
```

次の例を使用しましょう。

```
1  /* ex3 Sorting ints with qsort */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int comp(const void * _a, const void * _b)
7  {
8      const int *a=(const int *)_a;
9      const int *b=(const int *)_b;
10
11     if (*a==*b)
12         return 0;
13     else
14         if (*a < *b)
15             return -1;
16         else
17             return 1;
18 }
19
20 int main(int argc, char* argv[])
21 {
22     int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
23     int i;
24
25     /* Sort the array */
26     qsort(numbers,10,sizeof(int),comp) ;
27     for (i=0;i<9;i++)
28         printf("Number = %d\n",numbers[ i ]) ;
29     return 0;
30 }
```

第1.25.1節MSVC

MSVC 2010（簡潔にするため一部を省略）で /Ox オプションを付けてコンパイルしましょう。

Listing 1.359: 最適化 MSVC 2010: /GS- /MD

```
__a$ = 8 ; size = 4
```



```

    _b$ = 12 ; size = 4
_comp PROC
    mov     eax, DWORD PTR __a$[esp-4]
    mov     ecx, DWORD PTR __b$[esp-4]
    mov     eax, DWORD PTR [eax]
    mov     ecx, DWORD PTR [ecx]
    cmp     eax, ecx
    jne     SHORT $LN4@comp
    xor     eax, eax
    ret     0
$LN4@comp:
    xor     edx, edx
    cmp     eax, ecx
    setge   dl
    lea     eax, DWORD PTR [edx+edx-1]
    ret     0
_comp ENDP

_numbers$ = -40 ; size = 40
_argc$ = 8 ; size = 4
_argv$ = 12 ; size = 4
_main PROC
    sub     esp, 40 ; 00000028H
    push    esi
    push    OFFSET _comp
    push    4
    lea     eax, DWORD PTR _numbers$[esp+52]
    push    10 ; 0000000aH
    push    eax
    mov     DWORD PTR _numbers$[esp+60], 1892 ; 00000764H
    mov     DWORD PTR _numbers$[esp+64], 45 ; 0000002dH
    mov     DWORD PTR _numbers$[esp+68], 200 ; 000000c8H
    mov     DWORD PTR _numbers$[esp+72], -98 ; ffffffff9eH
    mov     DWORD PTR _numbers$[esp+76], 4087 ; 00000ff7H
    mov     DWORD PTR _numbers$[esp+80], 5
    mov     DWORD PTR _numbers$[esp+84], -12345 ; ffffcfc7H
    mov     DWORD PTR _numbers$[esp+88], 1087 ; 0000043fH
    mov     DWORD PTR _numbers$[esp+92], 88 ; 00000058H
    mov     DWORD PTR _numbers$[esp+96], -100000 ; fffe7960H
    call    _qsort
    add     esp, 16 ; 00000010H
...

```

これまでのところ驚くべきことは何也没有什么。4番目の引数として、ラベル `_comp` のアドレスが渡されます。これは `comp()` が置かれている場所、つまりその関数の最初の命令のアドレスです。

`qsort()` はどうやって呼び出すのでしょうか？

MSVCR80.DLL (C標準ライブラリ関数を含むMSVC DLLモジュール) にあるこの関数を見てください。

Listing 1.360: MSVCR80.DLL

```

.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, unsigned int, int
               (__cdecl *)(const void *, const void *))
.text:7816CBF0         public _qsort
.text:7816CBF0         _qsort
.text:7816CBF0         proc near
.text:7816CBF0 lo             = dword ptr -104h
.text:7816CBF0 hi             = dword ptr -100h
.text:7816CBF0 var_FC         = dword ptr -0FCh
.text:7816CBF0 stkp1r         = dword ptr -0F8h
.text:7816CBF0 lostk          = dword ptr -0F4h
.text:7816CBF0 histk          = dword ptr -7Ch
.text:7816CBF0 base           = dword ptr 4
.text:7816CBF0 num            = dword ptr 8
.text:7816CBF0 width          = dword ptr 0Ch
.text:7816CBF0 comp           = dword ptr 10h
.text:7816CBF0
.text:7816CBF0         sub     esp, 100h

....

.text:7816CCE0 loc_7816CCE0:                                     ; CODE XREF: _qsort+B1
.text:7816CCE0         shr     eax, 1
.text:7816CCE2         imul    eax, ebp
.text:7816CCE5         add     eax, ebx
.text:7816CCE7         mov     edi, eax
.text:7816CCE9         push    edi
.text:7816CCEA         push    ebx
.text:7816CCEB         call   [esp+118h+comp]
.text:7816CCF2         add     esp, 8
.text:7816CCF5         test    eax, eax
.text:7816CCF7         jle     short loc_7816CD04

```

comp は関数への4番目の引数です。ここで、制御は comp への引数のアドレスに渡されます。その前に、comp() に対して2つの引数が用意されています。その結果は実行後にチェックされます。

そのため、関数へのポインタを使用するのは危険です。まず第一に、誤った関数ポインタで qsort() を呼び出すと、qsort() が誤ったポイントに制御フローを渡すことがあり、プロセスがクラッシュしてこのバグを見つけるのが難しくなります。

2番目の理由は、コールバック関数の型は厳密に従わなければならない、間違っただけの引数で間違っただけの関数を呼び出すと深刻な問題につながる可能性があるということです。ただし、プロセスのクラッシュは問題ではありません、問題はクラッシュの理由をどうやって決定するのかです、なぜならコンパイラはコンパイル中に潜在的な問題について沈黙しているかもしれないからです。

MSVC + OllyDbg

例を OllyDbg にロードし、`comp()` にブレークポイントを設定しましょう。最初の `comp()` 呼び出しで値がどのように比較されるかを見ることができます。

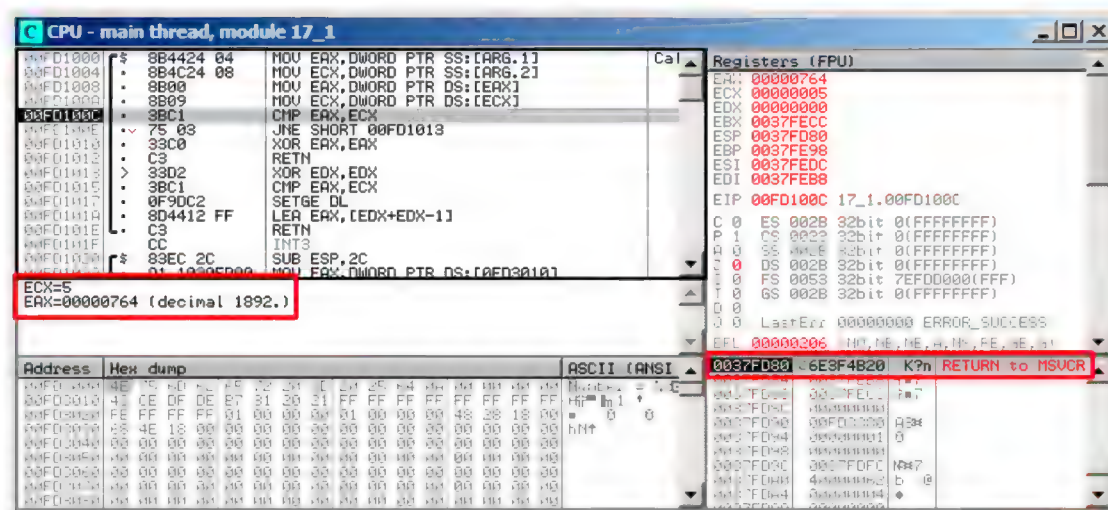


図 1.109: OllyDbg: `comp()` の最初の呼び出し

便宜上、OllyDbg はコードウィンドウの下ウィンドウに比較値を表示します。また、**SP!**が**RA**を指していることがわかります (`qsort()` 関数は `MSVC100.DLL` にあります)。

RETN 命令までトレースし (F8)、もう一度F8を押すと、qsort() 関数に戻ります。

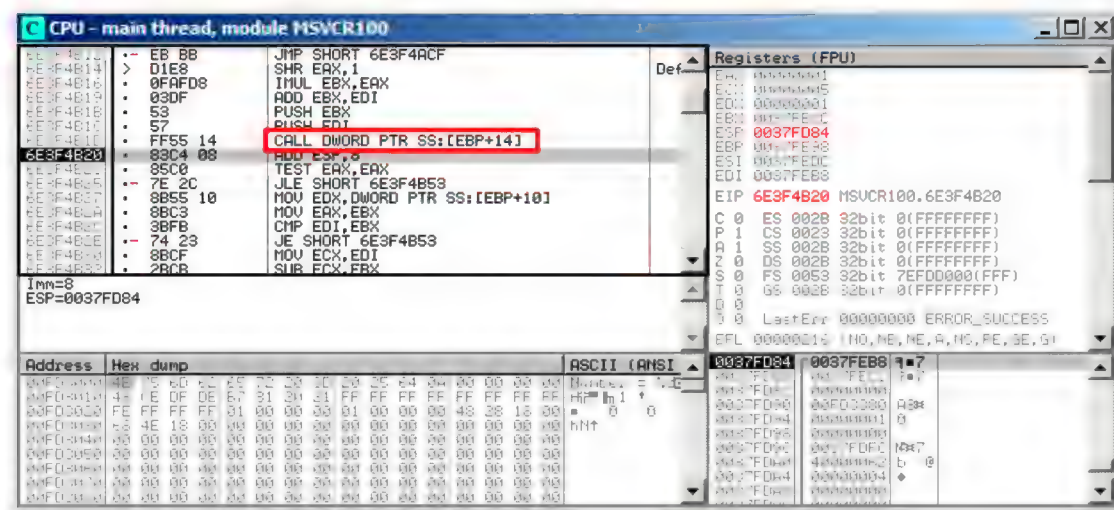


図 1.110: OllyDbg: comp() 呼び出し直後の qsort() のコード

それは比較関数への呼び出しでした。

これは `comp()` の2回目の呼び出しの瞬間のスクリーンショットでもあり、比較する必要がある値は異なります。

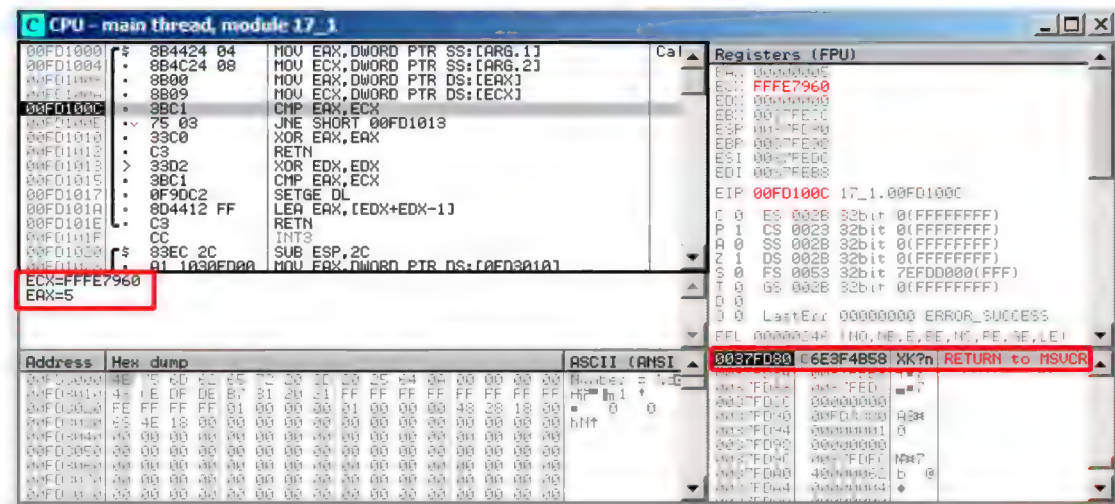


図 1.111: OllyDbg: `comp()` の2回目の呼び出し

MSVC + tracer

どのペアが比較されるかも見てみましょう。1892、45、200、-98、4087、5、-12345、1087、88、-100000の10個の番号がソートされています。

`comp()` で最初の `CMP` 命令のアドレスを取得しました。これは `0x0040100C` です。ここにブレークポイントを設定しました。

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

これで、ブレークポイントのレジスタに関する情報がいくつか得られます。

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xffff7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

EAX と ECX を除外すると以下のようになります。

```

EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0x0000ff7 ECX=0x00000005
EAX=0x0000ff7 ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x0000ff7
EAX=0x0000002d ECX=0x0000ff7
EAX=0x0000043f ECX=0x0000ff7
EAX=0x00000058 ECX=0x0000ff7
EAX=0x00000764 ECX=0x0000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058

```

34ペアです。したがって、クイックソートアルゴリズムでは、これら10個の数字をソートするために34回の比較操作が必要です。

MSVC + tracer (code coverage)

トレーサの機能を使用して可能なすべてのレジスタ値を収集し、それらを IDA に表示することもできます。

comp() 内のすべての命令をトレースしましょう。

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

IDA にロードするための.idc-script を入手してロードします。

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare proc near ; DATA XREF: _main+510
.text:00401000
.text:00401000 arg_0 = dword ptr 4
.text:00401000 arg_4 = dword ptr 8
.text:00401000
.text:00401000 mov eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004 mov ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008 mov eax, [eax] ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100a mov ecx, [ecx] ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xffff7960
.text:0040100c cmp eax, ecx ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100e jnz short loc_401013 ; ZF=false
.text:00401010 xor eax, eax
.text:00401012 ret
.text:00401013 ; -----
.text:00401013
.text:00401013 loc_401013: ; CODE XREF: PtFuncCompare+E1j
.text:00401013 xor edx, edx
.text:00401015 cmp eax, ecx ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017 setnl dl ; SF=false,true OF=false
.text:0040101a lea eax, [edx+edx-1]
.text:0040101e ret ; EAX=1, 0xffffffff
.text:0040101e PtFuncCompare endp
.text:0040101f
```

図 1.112: トレーサとIDA。注意：値がいくつか右側で切れています

IDA は関数に名前 (PtFuncCompare) を付けました。IDA はこの関数へのポインタが qsort() に渡されることを認識しているためです。

a ポインタと *b* ポインタは配列内のさまざまな場所を指していますが、32ビット値が配列に格納されているため、それらの間のステップは4です。

0x401010 と 0x401012 の命令は実行されなかったことがわかります（したがって、それらは白のままになります）。実際、comp() は0を返すことはありません。これは、配列内に等しい要素がないためです。

第1.25.2節GCC

大きな違いはありません。

Listing 1.361: GCC

```
lea    eax, [esp+40h+var_28]
mov     [esp+40h+var_40], eax
mov     [esp+40h+var_28], 764h
mov     [esp+40h+var_24], 2Dh
```

```

mov     [esp+40h+var_20], 0C8h
mov     [esp+40h+var_1C], 0FFFFFF9Eh
mov     [esp+40h+var_18], 0FF7h
mov     [esp+40h+var_14], 5
mov     [esp+40h+var_10], 0FFFCFC7h
mov     [esp+40h+var_C], 43Fh
mov     [esp+40h+var_8], 58h
mov     [esp+40h+var_4], 0FFE7960h
mov     [esp+40h+var_34], offset comp
mov     [esp+40h+var_38], 4
mov     [esp+40h+var_3C], 0Ah
call    _qsort

```

comp() 関数

```

comp     public comp
proc near

arg_0     = dword ptr 8
arg_4     = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
        mov     eax, [ebp+arg_4]
        mov     ecx, [ebp+arg_0]
        mov     edx, [eax]
        xor     eax, eax
        cmp     [ecx], edx
        jnz     short loc_8048458
        pop     ebp
        retn

loc_8048458:
        setnl   al
        movzx   eax, al
        lea     eax, [eax+eax-1]
        pop     ebp
        retn

comp     endp

```

qsort() の実装は libc.so.6 にあり、実際は単に qsort_r() のラッパーです ¹⁵⁸。

次に、quicksort() を呼び出しています。ここでは、定義済みの関数が渡されたポインタを介して呼び出されます。

Listing 1.362: (file libc.so.6, glibc version—2.10.1)

```

...
.text:0002DDF6      mov     edx, [ebp+arg_10]
.text:0002DDF9      mov     [esp+4], esi
.text:0002DDFD      mov     [esp], edi
.text:0002DE00      mov     [esp+8], edx
.text:0002DE04      call    [ebp+arg_C]

```

¹⁵⁸ [thunk function](#) と似たコンセプト

...

GCC + GDB (ソースコード付き)

明らかに、この例のCソースコード ([1.25 on page 468](#)) があるので、行番号 (11: 最初の比較が行われる行) にブレークポイント (*b*) を設定できます。また、デバッグ情報が含まれるように (-g) 例をコンパイルする必要があります。それで、アドレスと対応する行番号のテーブルが表示されます。

変数名 (*p*) を使って値を出力することもできます。デバッグ情報から、どのレジスタやローカルスタック要素にどの変数が含まれているかがわかります。

スタック (*bt*) を見て、Glibcで使用されている中間関数 `msort_with_tmp()` があることもわかります。

Listing 1.363: GDB session

```
dennis@ubuntuvml:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvml:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11      if (*a==*b)
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11      if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0
    ↪xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
    ↪.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=
    ↪5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort.c:45
    ↪.c:45
```

```
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↵
↳ =10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at ↵
↳ msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=↵
↳ cmp@entry=0x804844d <comp>,
   arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp↵
↳ >) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

GCC + GDB (ソースコードなし)

しかし多くの場合ソースコードが全くないので、`comp()` 関数を逆アセンブルし (`disas`)、一番最初の `CMP` 命令を見つけてそのアドレスにブレークポイント (`b`) を設定することができます。

各ブレークポイントで、すべてのレジスタの内容 (`info registers`) をダンプします。スタック情報も利用可能です (`bt`) が、

部分的です。`comp()` の行番号情報はありません。

Listing 1.364: GDB session

```
dennis@ubuntuv:~/polygon$ gcc 17_1.c
dennis@ubuntuv:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols ↵
↳ found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
   0x0804844d <+0>:      push    ebp
   0x0804844e <+1>:      mov     ebp,esp
   0x08048450 <+3>:      sub     esp,0x10
   0x08048453 <+6>:      mov     eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9>:      mov     DWORD PTR [ebp-0x8],eax
   0x08048459 <+12>:     mov     eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:     mov     DWORD PTR [ebp-0x4],eax
   0x0804845f <+18>:     mov     eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21>:     mov     edx,DWORD PTR [eax]
   0x08048464 <+23>:     mov     eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26>:     mov     eax,DWORD PTR [eax]
   0x08048469 <+28>:     cmp     edx,eax
   0x0804846b <+30>:     jne     0x8048474 <comp+39>
   0x0804846d <+32>:     mov     eax,0x0
   0x08048472 <+37>:     jmp     0x804848e <comp+65>
   0x08048474 <+39>:     mov     eax,DWORD PTR [ebp-0x8]
   0x08048477 <+42>:     mov     edx,DWORD PTR [eax]
   0x08048479 <+44>:     mov     eax,DWORD PTR [ebp-0x4]
```

```

0x0804847c <+47>:  mov    eax, DWORD PTR [eax]
0x0804847e <+49>:  cmp    edx, eax
0x08048480 <+51>:  jge    0x8048489 <comp+60>
0x08048482 <+53>:  mov    eax, 0xffffffff
0x08048487 <+58>:  jmp    0x804848e <comp+65>
0x08048489 <+60>:  mov    eax, 0x1
0x0804848e <+65>:  leave
0x0804848f <+66>:  ret

```

End of assembler dump.

(gdb) b *0x08048469

Breakpoint 1 at 0x8048469

(gdb) run

Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

```

eax          0x2d      45
ecx          0xbffff0f8  -1073745672
edx          0x764     1892
ebx          0xb7fc0000  -1208221696
esp          0xbffffeeb8 0xbffffeeb8
ebp          0xbffffeec8 0xbffffeec8
esi          0xbffff0fc  -1073745668
edi          0xbffff010  -1073745904
eip          0x8048469  0x8048469 <comp+28>
eflags      0x286     [ PF SF IF ]
cs          0x73      115
ss          0x7b      123
ds          0x7b      123
es          0x7b      123
fs          0x0       0
gs          0x33      51

```

(gdb) c

Continuing.

Breakpoint 1, 0x08048469 in comp ()

(gdb) info registers

```

eax          0xff7     4087
ecx          0xbffff104  -1073745660
edx          0xffffffff9e -98
ebx          0xb7fc0000  -1208221696
esp          0xbffffee58 0xbffffee58
ebp          0xbffffee68 0xbffffee68
esi          0xbffff108  -1073745656
edi          0xbffff010  -1073745904
eip          0x8048469  0x8048469 <comp+28>
eflags      0x282     [ SF IF ]
cs          0x73      115
ss          0x7b      123
ds          0x7b      123
es          0x7b      123
fs          0x0       0
gs          0x33      51

```

```

(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xffffffff9e      -98
ecx            0xbffff100      -1073745664
edx            0xc8            200
ebx            0xb7fc0000      -1208221696
esp            0xbffffeeb8     0xbffffeeb8
ebp            0xbffffeec8     0xbffffeec8
esi            0xbffff104      -1073745660
edi            0xbffff010      -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286          [ PF SF IF ]
cs             0x73            115
ss             0x7b            123
ds             0x7b            123
es             0x7b            123
fs             0x0             0
gs             0x33            51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0↵
    ↵ xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c) at msort↵
    ↵ .c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↵
    ↵ =5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c) at msort↵
    ↵ .c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry↵
    ↵ =10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c) at ↵
    ↵ msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=↵
    ↵ cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0x804844d <comp↵
    ↵ >) at msort.c:307
#9  0x0804850d in main ()

```

第1.25.3節関数へのポインタの危なさ

ご覧のとおり、`qsort()` 関数は、2つの `void*` 引数を取り、整数を返す関数へのポインタを期待しています。コード内に複数の比較関数がある場合（1つは文字列、もう1つは整数などを比較します）、それらを互いに混同することは非常に簡単です。あなたは整数を比較する関数を使って文字列の配列をソートしようとすることができます、そしてコンパイラはバグについてあなたに警告しません。

第1.26節32ビット環境での64ビット値

32ビット環境では、**GPR**は32ビットなので、64ビット値は32ビット値ペアとして格納され、渡されます。¹⁵⁹

第1.26.1節64ビットの値を返す

```
#include <stdint.h>

uint64_t f ()
{
    return 0x1234567890ABCDEF;
};
```

x86

32ビット環境では、64ビットの値は EDX:EAX レジスタペアを使って関数から返されます。

Listing 1.365: 最適化 MSVC 2010

```
_f      PROC
        mov     eax, -1867788817 ; 90abcdefH
        mov     edx, 305419896   ; 12345678H
        ret     0
_f      ENDP
```

ARM

64ビットの値は R0-R1 レジスタペアを使って返されます (R1 は高位の部分を R0 は低位の部分です)。

Listing 1.366: 最適化 Keil 6/2013 (ARMモード)

```
||f|| PROC
        LDR     r0, |L0.12|
        LDR     r1, |L0.16|
        BX      lr
        ENDP

|L0.12|
        DCD     0x90abcdef

|L0.16|
        DCD     0x12345678
```

MIPS

64ビットの値は V0-V1 (\$2-\$3) レジスタペアを使って返されます (V0 (\$2) は高位の部分を V1 (\$3) は低位の部分です)。

¹⁵⁹ちなみに、32ビット値は16ビット環境でも同様にペアとして渡されます：?? on page ??

Listing 1.367: 最適化 GCC 4.4.5 (assembly listing)

```

li    $3, -1867841536    # 0xffffffff90ab0000
li    $2, 305397760      # 0x12340000
ori   $3, $3, 0xcdef
j     $31
ori   $2, $2, 0x5678

```

Listing 1.368: 最適化 GCC 4.4.5 (IDA)

```

lui    $v1, 0x90AB
lui    $v0, 0x1234
li     $v1, 0x90ABCDEF
jr     $ra
li     $v0, 0x12345678

```

第1.26.2節 Arguments passing, addition, subtraction

```

#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
    return a+b;
};

void f_add_test ()
{
#ifdef __GNUC__
    printf ("%lld\n", f_add(12345678901234, 2345678901234));
#else
    printf ("%I64d\n", f_add(12345678901234, 2345678901234));
#endif
};

uint64_t f_sub (uint64_t a, uint64_t b)
{
    return a-b;
};

```

x86

Listing 1.369: 最適化 MSVC 2012 /Ob1

```

_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_f_add PROC
    mov     eax, DWORD PTR _a$[esp-4]
    add     eax, DWORD PTR _b$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    adc     edx, DWORD PTR _b$[esp]
    ret     0
_f_add ENDP

```

```

_f_add_test PROC
    push    5461                ; 00001555H
    push    1972608889         ; 75939f79H
    push    2874                ; 00000b3aH
    push    1942892530         ; 73ce2ff2H
    call    _f_add
    push    edx
    push    eax
    push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
    call    _printf
    add     esp, 28
    ret     0
_f_add_test ENDP

_f_sub PROC
    mov     eax, DWORD PTR _a$[esp-4]
    sub     eax, DWORD PTR _b$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    sbb     edx, DWORD PTR _b$[esp]
    ret     0
_f_sub ENDP

```

`f_add_test()` 関数では、各64ビット値が2つの32ビット値を使用して渡されることを確認できます。上位部分が最初に、次に下位部分になります。

足し算と引き算もペアで行われます。

さらに、下位32ビット部分が最初に追加されます。加算中にキャリーが発生した場合は、CF フラグが設定されます。

次の ADC 命令は、値の上位部分を加算し、 $CF = 1$ の場合は1を加算します。

減算もペアで行われます。最初の SUB は、後続の SBB 命令でチェックされるCFフラグをオンにすることもできます。キャリーフラグがオンの場合は、結果から1も減算されます。

`f_add()` 関数の結果がどのように `printf()` に渡されるのかを理解するのは簡単です。

Listing 1.370: GCC 4.8.1 -O1 -fno-inline

```

_f_add:
    mov     eax, DWORD PTR [esp+12]
    mov     edx, DWORD PTR [esp+16]
    add     eax, DWORD PTR [esp+4]
    adc     edx, DWORD PTR [esp+8]
    ret

_f_add_test:
    sub     esp, 28
    mov     DWORD PTR [esp+8], 1972608889 ; 75939f79H
    mov     DWORD PTR [esp+12], 5461      ; 00001555H
    mov     DWORD PTR [esp], 1942892530   ; 73ce2ff2H
    mov     DWORD PTR [esp+4], 2874       ; 00000b3aH
    call    _f_add
    mov     DWORD PTR [esp+4], eax

```

```

        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\n"
        call    _printf
        add     esp, 28
        ret

_f_sub:
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sub     edx, DWORD PTR [esp+16]
        ret

```

GCCのコードも同様です。

ARM

Listing 1.371: 最適化 Keil 6/2013 (ARMモード)

```

f_add PROC
    ADDS     r0,r0,r2
    ADC      r1,r1,r3
    BX       lr
    ENDP

f_sub PROC
    SUBS     r0,r0,r2
    SBC      r1,r1,r3
    BX       lr
    ENDP

f_add_test PROC
    PUSH     {r4,lr}
    LDR      r2,|L0.68| ; 0x75939f79
    LDR      r3,|L0.72| ; 0x00001555
    LDR      r0,|L0.76| ; 0x73ce2ff2
    LDR      r1,|L0.80| ; 0x00000b3a
    BL       f_add
    POP      {r4,lr}
    MOV      r2,r0
    MOV      r3,r1
    ADR      r0,|L0.84| ; "%I64d\n"
    B        __2printf
    ENDP

|L0.68|
DCD         0x75939f79
|L0.72|
DCD         0x00001555
|L0.76|
DCD         0x73ce2ff2
|L0.80|
DCD         0x00000b3a

```


L0.84	DCB	"%I64d\n",0
-------	-----	-------------

最初の64ビット値は R0 と R1 のレジスタペアに渡され、2番目の値は R2 と R3 のレジスタペアに渡されます。ARMには ADC 命令（キャリーフラグをカウントする）と SBC（「subtract with carry」）もあります。重要なこと：下位部分が加算/減算されるとき、-S接尾辞付きの ADDS および SUBS 命令が使用されます。-S接尾辞は「set flags」をあらわし、flags（特にキャリーフラグ）は、結果として生じる ADC/SBC 命令が確実に必要とするものです。そうでなければ、接尾辞-Sを付けずに命令を実行します（ADD および SUB）。

MIPS

Listing 1.372: 最適化 GCC 4.4.5 (IDA)

```
f_add:
; $a0 - high part of a
; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
        addu    $v1, $a3, $a1 ; sum up low parts
        addu    $a0, $a2, $a0 ; sum up high parts
; will carry generated while summing up low parts?
; if yes, set $v0 to 1
        sltu    $v0, $v1, $a3
        jr      $ra
; add 1 to high part of result if carry should be generated:
        addu    $v0, $a0 ; branch delay slot
; $v0 - high part of result
; $v1 - low part of result

f_sub:
; $a0 - high part of a
; $a1 - low part of a
; $a2 - high part of b
; $a3 - low part of b
        subu    $v1, $a1, $a3 ; subtract low parts
        subu    $v0, $a0, $a2 ; subtract high parts
; will carry generated while subtracting low parts?
; if yes, set $a0 to 1
        sltu    $a1, $v1
        jr      $ra
; subtract 1 from high part of result if carry should be generated:
        subu    $v0, $a1 ; branch delay slot
; $v0 - high part of result
; $v1 - low part of result

f_add_test:

var_10      = -0x10
var_4       = -4

        lui     $gp, (__gnu_local_gp >> 16)
        addiu   $sp, -0x20
```

```

        la    $gp, (__gnu_local_gp & 0xFFFF)
        sw    $ra, 0x20+var_4($sp)
        sw    $gp, 0x20+var_10($sp)
        lui   $a1, 0x73CE
        lui   $a3, 0x7593
        li    $a0, 0xB3A
        li    $a3, 0x75939F79
        li    $a2, 0x1555
        jal   f_add
        li    $a1, 0x73CE2FF2
        lw    $gp, 0x20+var_10($sp)
        lui   $a0, ($LC0 >> 16) # "%lld\n"
        lw    $t9, (printf & 0xFFFF)($gp)
        lw    $ra, 0x20+var_4($sp)
        la    $a0, ($LC0 & 0xFFFF) # "%lld\n"
        move  $a3, $v1
        move  $a2, $v0
        jr    $t9
        addiu $sp, 0x20

$LC0:    .ascii "%lld\n"<0>

```

MIPSにはフラグレジスタがないため、算術演算の実行後にそのような情報は存在しません。そのため、ADC や SBB ような命令はありません。キャリーフラグが設定されるかどうかを知るために、デスティネーションレジスタを1または0に設定する比較 {SLTU 命令を使用) も行われます。その後、この1または0が最終結果に加算または減算されます。

第1.26.3節乗算、除算

```

#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b)
{
    return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b)
{
    return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b)
{
    return a % b;
};

```

x86

Listing 1.373: 最適化 MSVC 2013 /Ob1

```

_a$ = 8 ; size = 8

```

```

_b$ = 16 ; size = 8
_f_mul PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __allmul ; long long multiplication
    pop     ebp
    ret     0
_f_mul ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_div PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __aulldiv ; unsigned long long division
    pop     ebp
    ret     0
_f_div ENDP

_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_rem PROC
    push    ebp
    mov     ebp, esp
    mov     eax, DWORD PTR _b$[ebp+4]
    push    eax
    mov     ecx, DWORD PTR _b$[ebp]
    push    ecx
    mov     edx, DWORD PTR _a$[ebp+4]
    push    edx
    mov     eax, DWORD PTR _a$[ebp]
    push    eax
    call    __aullrem ; unsigned long long remainder
    pop     ebp
    ret     0
_f_rem ENDP

```

乗算と除算はより複雑な演算なので、通常、コンパイラはそれを行うライブラリ関数への呼び出しを埋め込みます。

これらの機能はここに記述されています：?? on page ??

Listing 1.374: 最適化 GCC 4.8.1 -fno-inline

```

_f_mul:
    push    ebx
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+16]
    mov     ebx, DWORD PTR [esp+12]
    mov     ecx, DWORD PTR [esp+20]
    imul    ebx, eax
    imul    ecx, edx
    mul     edx
    add     ecx, ebx
    add     edx, ecx
    pop     ebx
    ret

_f_div:
    sub     esp, 28
    mov     eax, DWORD PTR [esp+40]
    mov     edx, DWORD PTR [esp+44]
    mov     DWORD PTR [esp+8], eax
    mov     eax, DWORD PTR [esp+32]
    mov     DWORD PTR [esp+12], edx
    mov     edx, DWORD PTR [esp+36]
    mov     DWORD PTR [esp], eax
    mov     DWORD PTR [esp+4], edx
    call    __udivdi3 ; unsigned division
    add     esp, 28
    ret

_f_rem:
    sub     esp, 28
    mov     eax, DWORD PTR [esp+40]
    mov     edx, DWORD PTR [esp+44]
    mov     DWORD PTR [esp+8], eax
    mov     eax, DWORD PTR [esp+32]
    mov     DWORD PTR [esp+12], edx
    mov     edx, DWORD PTR [esp+36]
    mov     DWORD PTR [esp], eax
    mov     DWORD PTR [esp+4], edx
    call    __umoddi3 ; unsigned modulo
    add     esp, 28
    ret

```

GCCは期待どおりに機能しますが、乗算コードは関数内でインライン化されているため、より効率的になる可能性があります。GCCには異なる関数名のライブラリあります：?? on page ??

ARM

ThumbモードのKeilはライブラリサブルーチン呼び出しを挿入します。

Listing 1.375: 最適化 Keil 6/2013 (Thumbモード)

```

||f_mul|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_lmul
    POP     {r4,pc}
    ENDP

||f_div|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_udivmod
    POP     {r4,pc}
    ENDP

||f_rem|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_udivmod
    MOVS    r0,r2
    MOVS    r1,r3
    POP     {r4,pc}
    ENDP

```

一方、ARMモードのKeilでは64ビットの乗算コードを生成できます。

Listing 1.376: 最適化 Keil 6/2013 (ARMモード)

```

||f_mul|| PROC
    PUSH    {r4,lr}
    UMULL   r12,r4,r0,r2
    MLA     r1,r2,r1,r4
    MLA     r1,r0,r3,r1
    MOV     r0,r12
    POP     {r4,pc}
    ENDP

||f_div|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_udivmod
    POP     {r4,pc}
    ENDP

||f_rem|| PROC
    PUSH    {r4,lr}
    BL      __aeabi_udivmod
    MOV     r0,r2
    MOV     r1,r3
    POP     {r4,pc}
    ENDP

```

MIPS

MIPS用に最適化 GCC 64ビット乗算コードを生成できますが、64ビット除算用のライブラリルーチンを読み出す必要があります。

Listing 1.377: 最適化 GCC 4.4.5 (IDA)

```
f_mul:
    mult    $a2, $a1
    mflo    $v0
    or      $at, $zero ; NOP
    or      $at, $zero ; NOP
    mult    $a0, $a3
    mflo    $a0
    addu    $v0, $a0
    or      $at, $zero ; NOP
    multu   $a3, $a1
    mfhi    $a2
    mflo    $v1
    jr      $ra
    addu    $v0, $a2

f_div:
var_10 = -0x10
var_4  = -4

    lui     $gp, (__gnu_local_gp >> 16)
    addiu   $sp, -0x20
    la      $gp, (__gnu_local_gp & 0xFFFF)
    sw      $ra, 0x20+var_4($sp)
    sw      $gp, 0x20+var_10($sp)
    lw      $t9, (__divdi3 & 0xFFFF)($gp)
    or      $at, $zero
    jalr    $t9
    or      $at, $zero
    lw      $ra, 0x20+var_4($sp)
    or      $at, $zero
    jr      $ra
    addiu   $sp, 0x20

f_rem:
var_10 = -0x10
var_4  = -4

    lui     $gp, (__gnu_local_gp >> 16)
    addiu   $sp, -0x20
    la      $gp, (__gnu_local_gp & 0xFFFF)
    sw      $ra, 0x20+var_4($sp)
    sw      $gp, 0x20+var_10($sp)
    lw      $t9, (__moddi3 & 0xFFFF)($gp)
    or      $at, $zero
    jalr    $t9
    or      $at, $zero
```

```

lw      $ra, 0x20+var_4($sp)
or       $at, $zero
jr       $ra
addiu    $sp, 0x20

```

たくさんのNOPがあります。おそらく乗算命令の後に埋められた遅延スロットです（結局のところ、それは他の命令より遅いです）。

第1.26.4節右シフト

```

#include <stdint.h>

uint64_t f (uint64_t a)
{
    return a>>7;
};

```

x86

Listing 1.378: 最適化 MSVC 2012 /Ob1

```

_a$ = 8      ; size = 8
_f          PROC
    mov     eax, DWORD PTR _a$[esp-4]
    mov     edx, DWORD PTR _a$[esp]
    shrd    eax, edx, 7
    shr     edx, 7
    ret     0
_f          ENDP

```

Listing 1.379: 最適化 GCC 4.8.1 -fno-inline

```

_f:
    mov     edx, DWORD PTR [esp+8]
    mov     eax, DWORD PTR [esp+4]
    shrd    eax, edx, 7
    shr     edx, 7
    ret

```

シフトは2つのパスでも発生します：最初に下部がシフトされ、次に上部がシフトされます。しかし、下位部分は SHRD 命令の助けを借りてシフトされ、それは EAX の値を7ビットだけシフトしますが、EDX からすなわち上位部分から新しいビットを引き出します。つまり、EDX:EAX レジスタのペアからの64ビット値は、全体として7ビットシフトされ、結果の最下位32ビットが EAX に格納されます。上位部分は、より一般的な SHR 命令を使用してシフトされます。実際、上位部分の解放されたビットはゼロで埋められなければなりません。

ARM

ARMはx86では SHRD のような命令を持っていないので、Keilコンパイラはこれを単純なシフトと OR 演算を使って行うべきです。

Listing 1.380: 最適化 Keil 6/2013 (ARMモード)

```
||f|| PROC
    LSR    r0,r0,#7
    ORR    r0,r0,r1,LSL #25
    LSR    r1,r1,#7
    BX     lr
    ENDP
```

Listing 1.381: 最適化 Keil 6/2013 (Thumbモード)

```
||f|| PROC
    LSLS   r2,r1,#25
    LSRS   r0,r0,#7
    ORRS   r0,r0,r2
    LSRS   r1,r1,#7
    BX     lr
    ENDP
```

MIPS

MIPS向けのGCCは、KeilがThumbモードで行うのと同じアルゴリズムに従います。

Listing 1.382: 最適化 GCC 4.4.5 (IDA)

```
f:
    sll    $v0, $a0, 25
    srl    $v1, $a1, 7
    or     $v1, $v0, $v1
    jr     $ra
    srl    $v0, $a0, 7
```

第1.26.5節32ビット値から64ビット値への変換

```
#include <stdint.h>

int64_t f (int32_t a)
{
    return a;
};
```

x86

Listing 1.383: 最適化 MSVC 2012

```
_a$ = 8
```



```

_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        cdq
        ret     0
_f      ENDP

```

ここでも、32ビットの符号付き値を64ビットの符号付き値に拡張する必要があります。符号なしの値は単純に変換されます：上位部分のすべてのビットは0に設定する必要があります。ただし、符号付きデータ型には適していません：符号は結果の数値の上位部分にコピーする必要があります。

CDQ 命令はここでそれを行います。EAX でその入力値を取り、それを64ビットに拡張しそして EDX:EAX レジスタペアに残します。つまり、CDQ は（EAXの最上位ビットを取得することによって）EAX から番号記号を取得し、それに応じて EDX の32ビットすべてを0または1に設定します。その動作は、MOVSX 命令とやや似ています。

ARM

Listing 1.384: 最適化 Keil 6/2013 (ARMモード)

```

||f|| PROC
        ASR     r1,r0,#31
        BX     lr
        ENDP

```

ARM用Keilは異なります。入力値を算術的に右に31ビットシフトします。知っての通り、符号ビットは**MSB**で、算術シフトは符号ビットを「出現した」ビットにコピーします。したがって、「ASR r1,r0,#31」の後、入力値が負の場合は R1 に0xFFFFFFFFが含まれ、それ以外の場合は0が含まれます。R1 には、結果の64ビット値の上位部分が含まれています。つまり、このコードは R0 の入力値から結果の64ビット値の上位32ビット部分のすべてのビットに**MSB**（符号ビット）をコピーするだけです。

MIPS

MIPS向けのGCCは、KeilがARMモードで行ったのと同じことを行います。

Listing 1.385: 最適化 GCC 4.4.5 (IDA)

```

f:
    sra     $v0, $a0, 31
    jr     $ra
    move    $v1, $a0

```

第1.27節SIMD

SIMD は頭字語です： *Single Instruction, Multiple Data*

名前の通り、複数のデータを1つの命令で処理します。

FPUと同様に、**CPU**サブシステムはx86内では独立したプロセッサのように見えます。

SIMDはx86でMMXとして始まりました。8つの新しい64ビットレジスタが登場しました：MM0-MM7

各MMXレジスタは、2つの32ビット値、4つの16ビット値、または8バイトを保持できます。たとえば、MMXレジスタに2つの値を追加することで、8つの8ビット値（バイト）を同時に追加することができます。

簡単な例として、画像を2次元配列として表現するグラフィックエディタがあります。ユーザが画像の明るさを変更すると、エディタは各ピクセル値に係数を加減する必要があります。簡潔にするために、画像がグレースケールで各ピクセルが1つの8ビットバイトで定義されているとしたら、8ピクセルの明るさを同時に変更することが可能です。

ところで、これが飽和命令がSIMDに存在する理由です。

ユーザがグラフィックエディタで明るさを変更するとき、オーバーフローとアンダーフローは望ましくないので、最大値に達すると何も加算しないという追加命令がSIMDにあります。

MMXが登場したとき、これらのレジスタは実際にはFPUのレジスタにありました。FPUまたはMMXを同時に使用することは可能でした。Intelはトランジスタを節約したと思うかもしれませんが、実際にはそのような共生の理由はより単純なものでした。追加のCPUレジスタを意識しないOSはコンテキストスイッチでそれらを保存せず、FPUレジスタを保存します。したがって、MMX機能を利用したMMX対応CPU + 古いOS + プロセスは依然として機能します。

SSEはSIMDレジスタを128ビットに拡張したもので、現在はFPUとは別のものです。

AVXは256ビットにした他の拡張です。

実用的な用途はどうでしょうか。

もちろん、これはメモリコピールーチン（memcpy）、メモリ比較（memcmp）などです。

もう1つの例：DES暗号化アルゴリズムは64ビットブロックと56ビットキーを受け取り、ブロックを暗号化して64ビットの結果を生成します。DESアルゴリズムは、ワイヤおよびAND/OR/NOTゲートを有する非常に大きな電子回路と見なすことができます。

Bitslice DES¹⁶⁰は、ブロックとキーのグループを同時に処理するというアイデアです。たとえば、x86の符号なし整数型の変数は最大32ビットを保持できるため、64個+56個の符号なし整数型の変数を使用して、32個のブロックキーペアの中間結果を同時に格納できます。

There is an utility to brute-force Oracle RDBMS passwords/hashes (ones based on DES), using slightly modified bitslice DES algorithm for SSE2 and AVX—now it is possible to encrypt 128 or 256 block-keys pairs simultaneously. SSE2およびAVX用にわずかに修正されたbitslice DESアルゴリズムを使用して、Oracle RDBMSのパスワード/ハッシュ（DESに基づくもの）をブルートフォースするユーティリティがあります。128または256のブロックキーペアを暗号化することが可能です。

http://conus.info/utils/ops_SIMD/

¹⁶⁰<http://www.darkside.com.au/bitslice/>

第1.27.1節ベクトル化

ベクトル化¹⁶¹は、たとえば、入力用に2つの配列を取り、1つの配列を生成するループがある場合です。ループ本体は入力配列から値を受け取り、何かを実行して結果を出力配列に入れます。ベクトル化は、いくつかの要素を同時に処理することです。

ベクトル化はそれほど新鮮なテクノロジーではありません。この教科書の作者は、少なくとも 1988年のCray Y-MPスーパーコンピュータラインで、その「ライト」バージョンのCray Y-MP EL 179を使ったことを見ました。¹⁶²

例えば：

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

このコード片は、AとBから要素を取り出し、それらを乗算して結果をCに保存します。

各配列要素が32ビット *int* の場合、Aから128ビットのXMMレジスタへ、Bから別のXMMレジスタへ、*PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*) および *PMULHW* (*Multiply Packed Signed Integers and Store High Result*) を実行することで、一度に4つの64ビット積^sを取得できます。

したがって、ループ本体の実行数は1024ではなく 1024/4 となり、4倍少なくなり、もちろん高速になります。

加算の例

Intel C++¹⁶³のように、単純な場合には自動的にベクトル化を実行できるコンパイラーもあります。

これが小さな機能です。

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];

    return 0;
};
```

Intel C++

それをIntel C++ 11.1.051 win32でコンパイルしましょう。

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

(IDA で) 次の結果を得ました。

¹⁶¹Wikipedia: [vectorization](#)

¹⁶²リモートから。それはスーパーコンピュータの博物館に設置されています: <http://www.cray-cyber.org>

¹⁶³インテルC++自動ベクトル化についての詳細: [Excerpt: Effective Automatic Vectorization](#)

```

; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz      = dword ptr  4
ar1     = dword ptr  8
ar2     = dword ptr 0Ch
ar3     = dword ptr 10h

        push    edi
        push    esi
        push    ebx
        push    esi
        mov     edx, [esp+10h+sz]
        test    edx, edx
        jle     loc_15B
        mov     eax, [esp+10h+ar3]
        cmp     edx, 6
        jle     loc_143
        cmp     eax, [esp+10h+ar2]
        jbe     short loc_36
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        neg     esi
        cmp     ecx, esi
        jbe     short loc_55

loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
        cmp     eax, [esp+10h+ar2]
        jnb     loc_143
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        cmp     esi, ecx
        jb      loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
        cmp     eax, [esp+10h+ar1]
        jbe     short loc_67
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        neg     esi
        cmp     ecx, esi
        jbe     short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
        cmp     eax, [esp+10h+ar1]
        jnb     loc_143
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        cmp     esi, ecx

```

```

        jb      loc_143
loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
        mov     edi, eax          ; edi = ar3
        and     edi, 0Fh         ; ar3は16バイト境界でアラインメントされているか?
        jz      short loc_9A     ; はい
        test    edi, 3
        jnz     loc_162
        neg     edi
        add     edi, 10h
        shr     edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
        lea     ecx, [edi+4]
        cmp     edx, ecx
        jl      loc_162
        mov     ecx, edx
        sub     ecx, edi
        and     ecx, 3
        neg     ecx
        add     ecx, edx
        test    edi, edi
        jbe     short loc_D6
        mov     ebx, [esp+10h+ar2]
        mov     [esp+10h+var_10], ecx
        mov     ecx, [esp+10h+ar1]
        xor     esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
        mov     edx, [ecx+esi*4]
        add     edx, [ebx+esi*4]
        mov     [eax+esi*4], edx
        inc     esi
        cmp     esi, edi
        jb      short loc_C1
        mov     ecx, [esp+10h+var_10]
        mov     edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
        mov     esi, [esp+10h+ar2]
        lea     esi, [esi+edi*4] ; ar2+i*4は16バイト境界でアラインメントされている
        か?
        test    esi, 0Fh
        jz      short loc_109    ; はい!
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
        movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
        movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4は16バイト境界でアラインメ
        ントされていないので、XMM0にロードされる
        paddb   xmm1, xmm0
        movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
        add     edi, 4

```

```

        cmp     edi, ecx
        jb      short loc_ED
        jmp      short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
        movdqu  xmm0, xmmword ptr [ebx+edi*4]
        paddb   xmm0, xmmword ptr [esi+edi*4]
        movdqa  xmmword ptr [eax+edi*4], xmm0
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
        ; f(int,int *,int *,int *)+164
        cmp     ecx, edx
        jnb      short loc_15B
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_133
        jmp      short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
        ; f(int,int *,int *,int *)+3A ...
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]
        xor     ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
        ; f(int,int *,int *,int *)+129 ...
        xor     eax, eax
        pop     ecx
        pop     ebx
        pop     esi
        pop     edi

```

```

    retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
        ; f(int,int *,int *,int *)+9F
        xor     ecx, ecx
        jmp     short loc_127
?f@@YAHHPAH00@Z endp

```

SSE2関連の命令は以下のとおりです。

- **MOVDQU (Move Unaligned Double Quadword)**—メモリから16バイトをXMMレジスタにロードします
- **PADDD (Add Packed Integers)**—4対の32ビット数を加算し、その結果を最初のオペランドに残します。ちなみに、オーバーフローが発生しても例外は発生せず、フラグも設定されません。結果の下位32ビットだけが格納されます。PADDD のオペランドの1つがメモリ内の値のアドレスである場合、そのアドレスは16バイト境界に揃えられている必要があります。整列されていない場合は、例外が発生します。
- **MOVDQA (Move Aligned Double Quadword)** はMOVDQUと同じですが、メモリ内の値のアドレスを16ビット境界に揃える必要があります。整列されていないと、例外が発生します。MOVDQA は MOVDQU よりも高速に動作しますが、前述のものがが必要です。

そのため、これらのSSE2命令は、作業するペアが4つ以上あり、ポインタ `ar3` が16バイト境界に整列している場合にのみ実行されます。

また、`ar2` が16バイト境界にも揃えられている場合は、次のコードが実行されます。

```

movdqu xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
padd    xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4

```

そうでなければ、`ar2` からの値は、MOVDQU を使用して XMM0 にロードされます。これは、位置合わせされたポインターを必要としませんが、遅くなる可能性があります。

```

movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4は16バイト境界にアラインメントされ
        ていないので、XMM0にロードされる
padd    xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4

```

それ以外の場合は、SSE2以外のコードが実行されます。

GCC

-O3 オプションが使用され、SSE2サポートがオンになっている場合、GCCは単純な場合にもベクトル化することがあります¹⁶⁴

以下を得ます (GCC 4.4.1)。

¹⁶⁴GCCベクトル化サポートについての詳細は: <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>

```

; f(int, int *, int *, int *)
      public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

      push    ebp
      mov     ebp, esp
      push    edi
      push    esi
      push    ebx
      sub     esp, 0Ch
      mov     ecx, [ebp+arg_0]
      mov     esi, [ebp+arg_4]
      mov     edi, [ebp+arg_8]
      mov     ebx, [ebp+arg_C]
      test    ecx, ecx
      jle     short loc_80484D8
      cmp     ecx, 6
      lea     eax, [ebx+10h]
      ja      short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
              ; f(int,int *,int *,int *)+61 ...
      xor     eax, eax
      nop
      lea     esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
      mov     edx, [edi+eax*4]
      add     edx, [esi+eax*4]
      mov     [ebx+eax*4], edx
      add     eax, 1
      cmp     eax, ecx
      jnz     short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
              ; f(int,int *,int *,int *)+A5
      add     esp, 0Ch
      xor     eax, eax
      pop     ebx
      pop     esi
      pop     edi
      pop     ebp
      retn

      align 8

```



```

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
            xor     edx, edx
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddb   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
            lea     edx, ds:0[eax*4]
            add     esi, edx
            add     edi, edx
            add     ebx, edx
            lea     esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
            mov     edx, [edi]
            add     eax, 1

```

```

        add     edi, 4
        add     edx, [esi]
        add     esi, 4
        mov     [ebx], edx
        add     ebx, 4
        cmp     ecx, eax
        jg      short loc_8048558
        add     esp, 0Ch
        xor     eax, eax
        pop     ebx
        pop     esi
        pop     edi
        pop     ebp
        retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
        cmp     eax, esi
        jnb     loc_80484C1
        jmp     loc_80484F8
_Z1fiPiS_S_ endp

```

しかし、ほぼ同じですが、Intel C++ほど細心の注意を払っていません。

メモリコピーの例

簡単なmemcpy() の例をもう一度見てみましょう。([1.16.2 on page 239](#)):

```

#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
};

```

GCC 4.9.1の最適化によるものです。

Listing 1.386: 最適化 GCC 4.9.1 x64

```

my_memcpy:
; RDI = コピー先アドレス
; RSI = コピー元アドレス
; RDX = ブロックサイズ
        test    rdx, rdx
        je      .L41
        lea     rax, [rdi+16]
        cmp     rsi, rax
        lea     rax, [rsi+16]
        setae   cl
        cmp     rdi, rax
        setae   al
        or      cl, al

```

```
je      .L13
cmp     rdx, 22
jbe     .L13
mov     rcx, rsi
push    rbp
push    rbx
neg     rcx
and     ecx, 15
cmp     rcx, rdx
cmova   rcx, rdx
xor     eax, eax
test    rcx, rcx
je      .L4
movzx   eax, BYTE PTR [rsi]
cmp     rcx, 1
mov     BYTE PTR [rdi], al
je      .L15
movzx   eax, BYTE PTR [rsi+1]
cmp     rcx, 2
mov     BYTE PTR [rdi+1], al
je      .L16
movzx   eax, BYTE PTR [rsi+2]
cmp     rcx, 3
mov     BYTE PTR [rdi+2], al
je      .L17
movzx   eax, BYTE PTR [rsi+3]
cmp     rcx, 4
mov     BYTE PTR [rdi+3], al
je      .L18
movzx   eax, BYTE PTR [rsi+4]
cmp     rcx, 5
mov     BYTE PTR [rdi+4], al
je      .L19
movzx   eax, BYTE PTR [rsi+5]
cmp     rcx, 6
mov     BYTE PTR [rdi+5], al
je      .L20
movzx   eax, BYTE PTR [rsi+6]
cmp     rcx, 7
mov     BYTE PTR [rdi+6], al
je      .L21
movzx   eax, BYTE PTR [rsi+7]
cmp     rcx, 8
mov     BYTE PTR [rdi+7], al
je      .L22
movzx   eax, BYTE PTR [rsi+8]
cmp     rcx, 9
mov     BYTE PTR [rdi+8], al
je      .L23
movzx   eax, BYTE PTR [rsi+9]
cmp     rcx, 10
mov     BYTE PTR [rdi+9], al
je      .L24
```

```

movzx    eax, BYTE PTR [rsi+10]
cmp      rcx, 11
mov      BYTE PTR [rdi+10], al
je       .L25
movzx    eax, BYTE PTR [rsi+11]
cmp      rcx, 12
mov      BYTE PTR [rdi+11], al
je       .L26
movzx    eax, BYTE PTR [rsi+12]
cmp      rcx, 13
mov      BYTE PTR [rdi+12], al
je       .L27
movzx    eax, BYTE PTR [rsi+13]
cmp      rcx, 15
mov      BYTE PTR [rdi+13], al
jne      .L28
movzx    eax, BYTE PTR [rsi+14]
mov      BYTE PTR [rdi+14], al
mov      eax, 15
.L4:
mov      r10, rdx
lea      r9, [rdx-1]
sub      r10, rcx
lea      r8, [r10-16]
sub      r9, rcx
shr      r8, 4
add      r8, 1
mov      r11, r8
sal      r11, 4
cmp      r9, 14
jbe      .L6
lea      rbp, [rsi+rcx]
xor      r9d, r9d
add      rcx, rdi
xor      ebx, ebx
.L7:
movdqa   xmm0, XMMWORD PTR [rbp+0+r9]
add      rbx, 1
movups   XMMWORD PTR [rcx+r9], xmm0
add      r9, 16
cmp      rbx, r8
jb       .L7
add      rax, r11
cmp      r10, r11
je       .L1
.L6:
movzx    ecx, BYTE PTR [rsi+rax]
mov      BYTE PTR [rdi+rax], cl
lea      rcx, [rax+1]
cmp      rdx, rcx
jbe      .L1
movzx    ecx, BYTE PTR [rsi+1+rax]
mov      BYTE PTR [rdi+1+rax], cl

```

```

lea    rcx, [rax+2]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+2+rax]
mov    BYTE PTR [rdi+2+rax], cl
lea    rcx, [rax+3]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+3+rax]
mov    BYTE PTR [rdi+3+rax], cl
lea    rcx, [rax+4]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+4+rax]
mov    BYTE PTR [rdi+4+rax], cl
lea    rcx, [rax+5]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+5+rax]
mov    BYTE PTR [rdi+5+rax], cl
lea    rcx, [rax+6]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+6+rax]
mov    BYTE PTR [rdi+6+rax], cl
lea    rcx, [rax+7]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+7+rax]
mov    BYTE PTR [rdi+7+rax], cl
lea    rcx, [rax+8]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+8+rax]
mov    BYTE PTR [rdi+8+rax], cl
lea    rcx, [rax+9]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+9+rax]
mov    BYTE PTR [rdi+9+rax], cl
lea    rcx, [rax+10]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+10+rax]
mov    BYTE PTR [rdi+10+rax], cl
lea    rcx, [rax+11]
cmp    rdx, rcx
jbe    .L1
movzx  ecx, BYTE PTR [rsi+11+rax]
mov    BYTE PTR [rdi+11+rax], cl
lea    rcx, [rax+12]
cmp    rdx, rcx
jbe    .L1

```

```

    movzx    ecx, BYTE PTR [rsi+12+rax]
    mov      BYTE PTR [rdi+12+rax], cl
    lea      rcx, [rax+13]
    cmp      rdx, rcx
    jbe      .L1
    movzx    ecx, BYTE PTR [rsi+13+rax]
    mov      BYTE PTR [rdi+13+rax], cl
    lea      rcx, [rax+14]
    cmp      rdx, rcx
    jbe      .L1
    movzx    edx, BYTE PTR [rsi+14+rax]
    mov      BYTE PTR [rdi+14+rax], dl
.L1:
    pop      rbx
    pop      rbp
.L41:
    rep ret
.L13:
    xor      eax, eax
.L3:
    movzx    ecx, BYTE PTR [rsi+rax]
    mov      BYTE PTR [rdi+rax], cl
    add      rax, 1
    cmp      rax, rdx
    jne      .L3
    rep ret
.L28:
    mov      eax, 14
    jmp      .L4
.L15:
    mov      eax, 1
    jmp      .L4
.L16:
    mov      eax, 2
    jmp      .L4
.L17:
    mov      eax, 3
    jmp      .L4
.L18:
    mov      eax, 4
    jmp      .L4
.L19:
    mov      eax, 5
    jmp      .L4
.L20:
    mov      eax, 6
    jmp      .L4
.L21:
    mov      eax, 7
    jmp      .L4
.L22:
    mov      eax, 8
    jmp      .L4

```

```

.L23:      mov     eax, 9
           jmp     .L4
.L24:      mov     eax, 10
           jmp     .L4
.L25:      mov     eax, 11
           jmp     .L4
.L26:      mov     eax, 12
           jmp     .L4
.L27:      mov     eax, 13
           jmp     .L4

```

第1.27.2節SIMD strlen() 実装

SIMD命令は、特別なマクロ¹⁶⁵を介して C/C++ コードに挿入できることに注意しなければなりません。MSVCの場合、それらのいくつかは `intrin.h` ファイルにあります。

SIMD命令を使用して `strlen()` 関数¹⁶⁶を実装することは、一般的な実装よりも2-2.5倍高速に実行できます。この関数は16文字をXMMレジスタにロードし、それぞれをゼロと照合します。¹⁶⁷

```

size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFFF0) == (unsigned int)↵
    ↵ str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;
        }
    }
}

```

¹⁶⁵MSDN: MMX, SSE, and SSE2 Intrinsics

¹⁶⁶`strlen()` — 文字列長を計算する標準Cライブラリ関数

¹⁶⁷この例は、以下のソースコードに基づいています。: http://www.strchr.com/sse2_optimised_strlen.

```

        break;
    }
    s += sizeof(__m128i);
    len += sizeof(__m128i);
};

return len;
}

```

/Ox オプションを付けてMSVC 2010でコンパイルしましょう。

Listing 1.387: 最適化 MSVC 2010

```

_pos$75552 = -4          ; サイズ = 4
_str$ = 8                ; サイズ = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push    ebp
    mov     ebp, esp
    and     esp, -16      ; ffffffff0H
    mov     eax, DWORD PTR _str$[ebp]
    sub     esp, 12       ; 0000000cH
    push    esi
    mov     esi, eax
    and     esi, -16      ; ffffffff0H
    xor     edx, edx
    mov     ecx, eax
    cmp     esi, eax
    je      SHORT $LN4@strlen_sse
    lea     edx, DWORD PTR [eax+1]
    npad    3 ; 次のラベルをアラインメント
$LL11@strlen_sse:
    mov     cl, BYTE PTR [eax]
    inc     eax
    test    cl, cl
    jne     SHORT $LL11@strlen_sse
    sub     eax, edx
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
$LN4@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [eax]
    pxor    xmm0, xmm0
    pcmpeqb xmm1, xmm0
    pmovmskb eax, xmm1
    test    eax, eax
    jne     SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [ecx+16]
    add     ecx, 16        ; 00000010H
    pcmpeqb xmm1, xmm0
    add     edx, 16        ; 00000010H
    pmovmskb eax, xmm1

```



```

    test    eax, eax
    je      SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf     eax, eax
    mov     ecx, eax
    mov     DWORD PTR _pos$75552[esp+16], eax
    lea     eax, DWORD PTR [ecx+edx]
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
?strlen_sse2@@YAIPBD@Z ENDP          ; strlen_sse2

```

どう機能するのでしょうか？まず第一に、私達は機能の目的を理解しなければなりません。これはC文字列の長さを計算しますが、別の用語を使用することもできます。タスクはゼロバイトを検索し、次に文字列の開始位置に対する位置を計算することです。

まず、str ポインタが16バイト境界に揃っているかどうかを調べます。そうでなければ、一般的な strlen() 実装を呼び出します。

次に、MOVDQA を使用して次の16バイトを XMM1 レジスタにロードします。

注意深い読者が尋ねるかもしれません、なぜポインターアライメントに関係なくメモリからデータをロードできるのに MOVDQU がここで使用できないのですか？

はい、それはこのようにして行われるかもしれません：もしポインタがアラインメントしていれば、MOVDQAを使用してデータをロードし、そうでなければより遅い MOVDQU を使用します。

しかし、ここで我々は別の警告を受けるかもしれません。

Windows NT系列の**OS**において（しかしそれに限定されない）、メモリは4KiB（4096バイト）のページによって割り当てられます。各win32プロセスには4GiBの空き容量がありますが、実際には、アドレス空間の一部のみが実際の物理メモリに接続されています。プロセスが存在しないメモリブロックにアクセスしている場合は、例外が発生します。それが**VM**のしくみです¹⁶⁸。

したがって、一度に16バイトをロードする関数は、割り当てられたメモリブロックの境界をまたぐことがあります。**OS**がアドレス0x008c0000に8192（0x2000）バイトを割り当てたとしましょう。したがって、ブロックはアドレス0x008c0000から始まり0x008c1fffまでを含むバイトです。

ブロックの後、つまりアドレス0x008c2000から始まり、そこには何もありません。**OS**はそこにメモリを割り当てていません。そのアドレスからメモリにアクセスしようとすると、例外が発生します。

プログラムがほぼブロックの最後に5文字を含む文字列を保持しているという例を考えてみましょう。それは違法な行為ではありません。

¹⁶⁸[wikipedia](#)

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	ランダムノイズ
0x008c1fff	ランダムノイズ

したがって、通常の状態では、プログラムは `strlen()` を呼び出して、アドレス `0x008c1ff8` のメモリに配置された文字列 `'hello'` へのポインタを渡します。`strlen()` は `0x008c1ffd` まで1バイトずつ読み込みます。`0x008c1ffd` はバイトが0ですが、その後は停止します。

整列されているかどうかにかかわらず、任意のアドレスから始めて一度に16バイトを読み取る独自の `strlen()` を実装すると、`MOVDQU` はアドレス `0x008c1ff8` から最大16バイトを一度に `0x008c2008` までロードしようとして、例外が発生します。もちろん、そのような状況は避けるべきです。

そのため、私たちは16バイト境界に整列されたアドレスでのみ動作します。これは、OSのページサイズが通常16バイト境界に整列されているという知識と組み合わせると、ある程度の保証が得られます。私たちの関数は、割り当てられていないメモリから読み込みません。

私たちの機能に戻りましょう。

`_mm_setzero_si128()` は、`pxor xmm0, xmm0` .itを生成するマクロで、XMM0レジスタをクリアするだけです。

`_mm_load_si128()` は `MOVDQA` のマクロで、アドレスからXMM1レジスタに16バイトをロードするだけです。

`_mm_cmpeq_epi8()` は、2つのXMMレジスタをバイト単位で比較する命令である `PCMPEQB` 用のマクロです。

また、あるバイトが他のレジスタのバイトと等しい場合は、結果のこの時点で `0xff` になり、それ以外の場合は0になります。

例えば：

```
XMM1: 0x11223344556677880000000000000000
XMM0: 0x11ab3444007877881111111111111111
```

`pcmpeqb xmm1, xmm0` の実行後、XMM1 レジスタには以下が含まれます。

```
XMM1: 0xffff0000ff0000ffff0000000000000000
```

この例では、この命令は各16バイトブロックを16個のゼロバイトのブロックと比較します。これは、`pxor xmm0, xmm0` によって XMM0 レジスタに設定されています。

次のマクロは `_mm_movemask_epi8()` です。これは `PMOVMSKB` 命令です。

`PCMPEQB` と一緒に使うととても便利です。

```
pmovmskb eax, xmm1
```

この命令は、XMM1 の最初のバイトの最上位ビットが1の場合、最初のEAXビットを1に設定します。つまり、XMM1 レジスタの最初のバイトが `0xff` の場合、EAX の最初のビット

も1になります。

XMM1 レジスタの2番目のバイトが 0xff の場合、EAX の2番目のビットは1に設定されます。言い換えれば、命令は「XMM1 のどのバイトに最上位ビット (MBS) が設定されているか、0x7fより大きいか」という質問に答えます。そして、EAX レジスタに16ビットを返します。EAX レジスタの他のビットはクリアされます。

ところで、私たちのアルゴリズムのこの風変わりなことを忘れないでください。入力には16バイトあります。

15	14	13	12	11	10	9		3	2	1	0
'h'	'e'	'l'	'l'	'o'	0		ガーベッジ		0		ガーベッジ

これは、'hello' 文字列で、ゼロで終わり、メモリ内のランダムノイズです。

これらの16バイトを XMM1 にロードしてゼロ化された XMM0 と比較すると、次のようになります。¹⁶⁹

XMM1: 0x0000ff0000000000000000ff0000000000

これは、命令が2つのゼロバイトを見つけたことを意味していますが、それは驚くことではありません。

この場合の PMOVMASKB は EAX を 0b0010000000100000 に設定します。

明らかに、私たちの関数は最初の0ビットだけを取り、残りを無視しなければなりません。

次の命令は BSF (*Bit Scan Forward*) です。

この命令は、1に設定された最初のビットを見つけ、その位置を最初のオペランドに格納します。

EAX=0b0010000000100000

bsf eax, eax の実行後、EAX は5を含み、これは1が5番目のビット位置（ゼロから始まる）に見つかったことを意味します。

MSVCには、この命令用のマクロ `_BitScanForward` があります。

今は簡単です。ゼロバイトが見つかった場合は、その位置がすでに数えたものに追加され、今度は結果が返されます。

Almost all. ほとんど全て。

ちなみに、MSVCコンパイラは最適化のために2つのループ本体を一緒に発行していました。

ちなみに、SSE 4.2 (Intel Core i7に登場) は、これらの文字列操作がさらに簡単になる可能性がある場合に、より多くの命令を提供します：http://www.strchr.com/strcmp_and_strlen_using_sse_4.2

¹⁶⁹ここでは、MSBからLSB¹⁷⁰への順序が使用されています。

第1.28節64ビット

第1.28.1節x86-64

これはx86アーキテクチャの64ビット拡張です。

リバースエンジニアの観点からすると、最も重要な変更は次のとおりです。

- ほとんどすべてのレジスタ（FPUとSIMDを除く）は64ビットに拡張され、Rプレフィックスが付けられました。8つの追加レジスタが追加されました。GPRは RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15 です。

従来通りに古いレジスタ部分にアクセスすることはまだ可能です。例えば、EAX を使用して RAX レジスタの下位32ビット部分にアクセスすることは可能です。

バイトの並び順							
第7	第6	第5	第4	第3	第2	第1	第0
RAX ^{x64}							
				EAX			
						AX	
						AH	AL

新しい R8-R15 レジスタの下位部分も、R8D-R15D（下位32ビット部分）、R8W-R15W（下位16ビット部分）、R8L-R15L（下位8ビット部分）です。

バイトの並び順							
第7	第6	第5	第4	第3	第2	第1	第0
R8							
				R8D			
						R8W	
						R8L	

SIMDレジスタの数は8から16に倍増しました：XMM0-XMM15

- Win64では、関数呼び出し規約は多少異なり、ややfastcallに似ています (?? on page ??)。最初の4つの引数は RCX、RDX、R8、R9 レジスタに格納され、残りはスタックに格納されます。また、caller関数は32バイトを割り当てなければならないので、calleeはそこに4つの最初の引数を保存し、それ自体の必要性のためにこれらのレジスタを使用することができます。短い関数は単にレジスタからの引数を使用するかもしれませんが、大きいものはそれらの値をスタックに保存するかもしれません。

System V AMD64 ABI (Linux、*BSD、Mac OS X) [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] ¹⁷¹はfastcallにも似ていますが、最初の6つの引数に6つのレジスタ RDI、RSI、RDX、RCX、R8、R9 を使用します。残りはすべてスタックを介して渡されます。

呼び出し規約 (?? on page ??) に関するセクションも参照してください。

- 互換性のため、C/C++ の *int* 型は32ビットのままです。
- ポインタはすべて64ビットです。

¹⁷¹以下で利用可能 <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

現在、レジスタの数が2倍になったため、コンパイラは[register allocation](#)と呼ばれる操作のためのスペースをより多く持っています。私たちにとってこれは、発行されたコードがより少ない数のローカル変数を含んでいることを意味します。

たとえば、DES暗号化アルゴリズムの最初のSボックスを計算する関数は、ビットスライスのDESメソッドを使用して (DES_type 型 (uint32、uint64、SSE2、またはAVXに応じて)) 32/64/128/256の値を一度に処理します。(このテクニックの詳細は[ここに](#)あります (1.27 on page 494))。

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
```

```
x10 = x7 ^ x9;
x11 = a2 | x10;
x12 = x6 ^ x11;
x13 = a5 ^ x5;
x14 = x13 & x8;
x15 = a5 & ~a4;
x16 = x3 ^ x14;
x17 = a6 | x16;
x18 = x15 ^ x17;
x19 = a2 | x18;
x20 = x14 ^ x19;
x21 = a1 & x20;
x22 = x12 ^ ~x21;
*out2 ^= x22;
x23 = x1 | x5;
x24 = x23 ^ x8;
x25 = x18 & ~x2;
x26 = a2 & ~x25;
x27 = x24 ^ x26;
x28 = x6 | x7;
x29 = x28 ^ x25;
x30 = x9 ^ x24;
x31 = x18 & ~x30;
x32 = a2 & x31;
x33 = x29 ^ x32;
x34 = a1 & x33;
x35 = x27 ^ x34;
*out4 ^= x35;
x36 = a3 & x28;
x37 = x18 & ~x36;
x38 = a2 | x3;
x39 = x37 ^ x38;
x40 = a3 | x31;
x41 = x24 & ~x37;
x42 = x41 | x3;
x43 = x42 & ~a2;
x44 = x40 ^ x43;
x45 = a1 & ~x44;
x46 = x39 ^ ~x45;
*out1 ^= x46;
x47 = x33 & ~x9;
x48 = x47 ^ x39;
x49 = x4 ^ x36;
x50 = x49 & ~x5;
x51 = x42 | x18;
x52 = x51 ^ a5;
x53 = a2 & ~x52;
x54 = x50 ^ x53;
x55 = a1 | x54;
x56 = x48 ^ ~x55;
*out3 ^= x56;
}
```

ローカル変数がたくさんあります。もちろん、すべてがローカルスタックに入るわけではありません。/Ox オプションを付けてMSVC 2008でコンパイルしてみましょう。

Listing 1.388: 最適化 MSVC 2008

```

PUBLIC      _s1
; Function compile flags: /Ogtpy
_TEXT      SEGMENT
_x6$ = -20      ; size = 4
_x3$ = -16      ; size = 4
_x1$ = -12      ; size = 4
_x8$ = -8       ; size = 4
_x4$ = -4       ; size = 4
_a1$ = 8        ; size = 4
_a2$ = 12       ; size = 4
_a3$ = 16       ; size = 4
_x33$ = 20      ; size = 4
_x7$ = 20       ; size = 4
_a4$ = 20       ; size = 4
_a5$ = 24       ; size = 4
tv326 = 28      ; size = 4
_x36$ = 28      ; size = 4
_x28$ = 28      ; size = 4
_a6$ = 28       ; size = 4
_out1$ = 32     ; size = 4
_x24$ = 36      ; size = 4
_out2$ = 36     ; size = 4
_out3$ = 40     ; size = 4
_out4$ = 44     ; size = 4
_s1        PROC
    sub     esp, 20      ; 00000014H
    mov     edx, DWORD PTR _a5$[esp+16]
    push    ebx
    mov     ebx, DWORD PTR _a4$[esp+20]
    push    ebp
    push    esi
    mov     esi, DWORD PTR _a3$[esp+28]
    push    edi
    mov     edi, ebx
    not     edi
    mov     ebp, edi
    and     edi, DWORD PTR _a5$[esp+32]
    mov     ecx, edx
    not     ecx
    and     ebp, esi
    mov     eax, ecx
    and     eax, esi
    and     ecx, ebx
    mov     DWORD PTR _x1$[esp+36], eax
    xor     eax, ebx
    mov     esi, ebp
    or      esi, edx
    mov     DWORD PTR _x4$[esp+36], esi
    and     esi, DWORD PTR _a6$[esp+32]

```

```

mov     DWORD PTR _x7$[esp+32], ecx
mov     edx, esi
xor     edx, eax
mov     DWORD PTR _x6$[esp+36], edx
mov     edx, DWORD PTR _a3$[esp+32]
xor     edx, ebx
mov     ebx, esi
xor     ebx, DWORD PTR _a5$[esp+32]
mov     DWORD PTR _x8$[esp+36], edx
and     ebx, edx
mov     ecx, edx
mov     edx, ebx
xor     edx, ebp
or      edx, DWORD PTR _a6$[esp+32]
not     ecx
and     ecx, DWORD PTR _a6$[esp+32]
xor     edx, edi
mov     edi, edx
or      edi, DWORD PTR _a2$[esp+32]
mov     DWORD PTR _x3$[esp+36], ebp
mov     ebp, DWORD PTR _a2$[esp+32]
xor     edi, ebx
and     edi, DWORD PTR _a1$[esp+32]
mov     ebx, ecx
xor     ebx, DWORD PTR _x7$[esp+32]
not     edi
or      ebx, ebp
xor     edi, ebx
mov     ebx, edi
mov     edi, DWORD PTR _out2$[esp+32]
xor     ebx, DWORD PTR [edi]
not     eax
xor     ebx, DWORD PTR _x6$[esp+36]
and     eax, edx
mov     DWORD PTR [edi], ebx
mov     ebx, DWORD PTR _x7$[esp+32]
or      ebx, DWORD PTR _x6$[esp+36]
mov     edi, esi
or      edi, DWORD PTR _x1$[esp+36]
mov     DWORD PTR _x28$[esp+32], ebx
xor     edi, DWORD PTR _x8$[esp+36]
mov     DWORD PTR _x24$[esp+32], edi
xor     edi, ecx
not     edi
and     edi, edx
mov     ebx, edi
and     ebx, ebp
xor     ebx, DWORD PTR _x28$[esp+32]
xor     ebx, eax
not     eax
mov     DWORD PTR _x33$[esp+32], ebx
and     ebx, DWORD PTR _a1$[esp+32]
and     eax, ebp

```



```

xor     eax, ebx
mov     ebx, DWORD PTR _out4$[esp+32]
xor     eax, DWORD PTR [ebx]
xor     eax, DWORD PTR _x24$[esp+32]
mov     DWORD PTR [ebx], eax
mov     eax, DWORD PTR _x28$[esp+32]
and     eax, DWORD PTR _a3$[esp+32]
mov     ebx, DWORD PTR _x3$[esp+36]
or      edi, DWORD PTR _a3$[esp+32]
mov     DWORD PTR _x36$[esp+32], eax
not     eax
and     eax, edx
or      ebx, ebp
xor     ebx, eax
not     eax
and     eax, DWORD PTR _x24$[esp+32]
not     ebp
or      eax, DWORD PTR _x3$[esp+36]
not     esi
and     ebp, eax
or      eax, edx
xor     eax, DWORD PTR _a5$[esp+32]
mov     edx, DWORD PTR _x36$[esp+32]
xor     edx, DWORD PTR _x4$[esp+36]
xor     ebp, edi
mov     edi, DWORD PTR _out1$[esp+32]
not     eax
and     eax, DWORD PTR _a2$[esp+32]
not     ebp
and     ebp, DWORD PTR _a1$[esp+32]
and     edx, esi
xor     eax, edx
or      eax, DWORD PTR _a1$[esp+32]
not     ebp
xor     ebp, DWORD PTR [edi]
not     ecx
and     ecx, DWORD PTR _x33$[esp+32]
xor     ebp, ebx
not     eax
mov     DWORD PTR [edi], ebp
xor     eax, ecx
mov     ecx, DWORD PTR _out3$[esp+32]
xor     eax, DWORD PTR [ecx]
pop     edi
pop     esi
xor     eax, ebx
pop     ebp
mov     DWORD PTR [ecx], eax
pop     ebx
add     esp, 20
ret     0
_s1     ENDP

```

5つの変数がコンパイラによってローカルスタックに割り当てられました。

それでは、64ビット版のMSVC 2008でも同じことを試してみましょう。

Listing 1.389: 最適化 MSVC 2008

```

a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1      PROC
$LN3:
    mov     QWORD PTR [rsp+24], rbx
    mov     QWORD PTR [rsp+32], rbp
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    push    rsi
    push    rdi
    push    r12
    push    r13
    push    r14
    push    r15
    mov     r15, QWORD PTR a5$[rsp]
    mov     rcx, QWORD PTR a6$[rsp]
    mov     rbp, r8
    mov     r10, r9
    mov     rax, r15
    mov     rdx, rbp
    not     rax
    xor     rdx, r9
    not     r10
    mov     r11, rax
    and     rax, r9
    mov     rsi, r10
    mov     QWORD PTR x36$1$[rsp], rax
    and     r11, r8
    and     rsi, r8
    and     r10, r15
    mov     r13, rdx
    mov     rbx, r11
    xor     rbx, r9
    mov     r9, QWORD PTR a2$[rsp]
    mov     r12, rsi
    or      r12, r15
    not     r13
    and     r13, rcx
    mov     r14, r12
    and     r14, rcx

```

```

mov     rax, r14
mov     r8, r14
xor     r8, rbx
xor     rax, r15
not     rbx
and     rax, rdx
mov     rdi, rax
xor     rdi, rsi
or      rdi, rcx
xor     rdi, r10
and     rbx, rdi
mov     rcx, rdi
or      rcx, r9
xor     rcx, rax
mov     rax, r13
xor     rax, QWORD PTR x36$1$(rsp)
and     rcx, QWORD PTR a1$(rsp)
or      rax, r9
not     rcx
xor     rcx, rax
mov     rax, QWORD PTR out2$(rsp)
xor     rcx, QWORD PTR [rax]
xor     rcx, r8
mov     QWORD PTR [rax], rcx
mov     rax, QWORD PTR x36$1$(rsp)
mov     rcx, r14
or      rax, r8
or      rcx, r11
mov     r11, r9
xor     rcx, rdx
mov     QWORD PTR x36$1$(rsp), rax
mov     r8, rsi
mov     rdx, rcx
xor     rdx, r13
not     rdx
and     rdx, rdi
mov     r10, rdx
and     r10, r9
xor     r10, rax
xor     r10, rbx
not     rbx
and     rbx, r9
mov     rax, r10
and     rax, QWORD PTR a1$(rsp)
xor     rbx, rax
mov     rax, QWORD PTR out4$(rsp)
xor     rbx, QWORD PTR [rax]
xor     rbx, rcx
mov     QWORD PTR [rax], rbx
mov     rbx, QWORD PTR x36$1$(rsp)
and     rbx, rbp
mov     r9, rbx
not     r9

```

```

and    r9, rdi
or     r8, r11
mov    rax, QWORD PTR out1$[rsp]
xor    r8, r9
not    r9
and    r9, rcx
or     rdx, rbp
mov    rbp, QWORD PTR [rsp+80]
or     r9, rsi
xor    rbx, r12
mov    rcx, r11
not    rcx
not    r14
not    r13
and    rcx, r9
or     r9, rdi
and    rbx, r14
xor    r9, r15
xor    rcx, rdx
mov    rdx, QWORD PTR a1$[rsp]
not    r9
not    rcx
and    r13, r10
and    r9, r11
and    rcx, rdx
xor    r9, rbx
mov    rbx, QWORD PTR [rsp+72]
not    rcx
xor    rcx, QWORD PTR [rax]
or     r9, rdx
not    r9
xor    rcx, r8
mov    QWORD PTR [rax], rcx
mov    rax, QWORD PTR out3$[rsp]
xor    r9, r13
xor    r9, QWORD PTR [rax]
xor    r9, r8
mov    QWORD PTR [rax], r9
pop    r15
pop    r14
pop    r13
pop    r12
pop    rdi
pop    rsi
ret    0
s1     ENDP

```

ローカルスタックにコンパイラによって割り当てられたものは何もあります。x36 は a5 と同義です。

ところで、もっと多くのGPRを持つCPUがあります。例えば、Itanium (128レジスタ) です。

第1.28.2節ARM

64ビット命令はARMv8で登場しました。

第1.28.3節Float point numbers

x86-64で浮動小数点数がどのように処理されるかは以下で説明されます：[1.29](#)

第1.28.4節64-bit architecture criticism

x64 CPUは48ビットの外部RAMしかアドレス指定できないという事実にもかかわらず、キャッシュメモリを含め、ポインタを格納するために2倍のメモリが必要になることに、いら立つ人がいます。

ここにある私の64ビットコンピュータでは、私が自分のマシンの能力を最大限使用することを気にしているのであれば、ポインタを使わない方がいいと思います。私は64ビットのレジスタを持つマシンを持っていますが、RAMは2ギガバイトしかありません。そのため、ポインタは32を超える有効ビットを持ちません。しかし、ポインタを使用するたびに64ビットのコストがかかり、データ構造のサイズが2倍になります。さらに悪いことに、ポインタはキャッシュに入り、私のキャッシュの半分がなくなり、キャッシュをキャッシュするコストは高くなります。

だから本当にエンベロープを押し込もうとするなら、私はポインタの代わりに配列を使わなければなりません。ポインタを使用しているように見えるように複雑なマクロを作成しますが、実際は使ってはいません。

(Donald Knuth in “Coders at Work: Reflections on the Craft of Programming ”.)

自分自身のメモリアロケータを作る人もいます。CryptoMiniSat¹⁷²のケースについて知っておくと面白いです。このプログラムはめったに4GiBを超えるRAMを使用しませんが、ポインタを頻繁に使用します。そのため、32ビットアーキテクチャでは64ビットアーキテクチャよりもメモリが少なく済みます。この問題を軽減するために、著者は独自のアロケータ (*clauseallocator.h|cpp*) ファイルに)を作成しました。これにより、64ビットポインタの代わりに32ビット識別子を使用して割り当てられたメモリにアクセスできます。

第1.29節SIMDを使用した浮動小数点数の取り扱い

もちろん、SIMD拡張機能が追加されたとき、FPUはx86互換プロセッサに残っていました。

SIMD拡張 (SSE2) は、浮動小数点数を扱うためのより簡単な方法を提供します。

数値のフォーマットは変わりません (IEEE 754)。

そのため、現代のコンパイラ (x86-64用に生成されたものも含む) は通常、FPUの代わりにSIMD命令を使用します。

彼らと一緒に動作する方が簡単なので、それは良いニュースだと言えます。

¹⁷²<https://github.com/msoos/cryptominisat/>

ここではFPUセクションの例を再利用します：[1.19 on page 268](#)

第1.29.1節単純な例

```
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

x64

Listing 1.390: 最適化 MSVC 2012 x64

```
__real@4010666666666666 DQ 0401066666666666r    ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14

a$ = 8
b$ = 16
f      PROC
        divsd    xmm0, QWORD PTR __real@40091eb851eb851f
        mulsd    xmm1, QWORD PTR __real@4010666666666666
        addsd    xmm0, xmm1
        ret      0
f      ENDP
```

入力浮動小数点値は XMM0-XMM3 レジスタに渡され、残りはすべてスタックを介して渡されます。¹⁷³

a は XMM0 に渡され、 b は XMM1 を介して渡されます。

XMMレジスタは128ビットです（SIMDに関するセクションからわかるように：[1.27 on page 493](#)）。しかし *double* 値は64ビットですので、下位半分のレジスタだけが使用されます。

DIVSD は「Divide Scalar Double-Precision Floating-Point Values」を表すSSE命令です。これは、オペランドの下半分に格納された *double* 型の値を別の値で除算するだけです。

定数は、コンパイラによってIEEE 754形式でエンコードされています。

MULSD と ADDSD はまったく同じように機能しますが、乗算と加算を行います。

関数が *double* 型で実行された結果は、XMM0 レジスタに残ります。

これが、最適化されていないMSVCの仕組みです。

¹⁷³[MSDN: Parameter Passing](#)

Listing 1.391: MSVC 2012 x64

```

__real@4010666666666666 DQ 0401066666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14

a$ = 8
b$ = 16
f
    PROC
        movsdx QWORD PTR [rsp+16], xmm1
        movsdx QWORD PTR [rsp+8], xmm0
        movsdx xmm0, QWORD PTR a$[rsp]
        divsd  xmm0, QWORD PTR __real@40091eb851eb851f
        movsdx xmm1, QWORD PTR b$[rsp]
        mulsd  xmm1, QWORD PTR __real@4010666666666666
        addsd  xmm0, xmm1
        ret    0
    f
    ENDP

```

少し冗長です。入力引数は「シャドースペース」(1.10.2 on page 126) に保存されますが、それらの下位レジスタのみが半分になります。つまり、*double* 型の64ビット値だけです。GCCは同じコードを生成します。

x86

この例もx86用にコンパイルしましょう。x86用に生成されているという事実にもかかわらず、MSVC 2012はSSE2命令を使用します。

Listing 1.392: 非最適化 MSVC 2012 x86

```

tv70 = -8 ; size = 8
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f
    PROC
        push ebp
        mov ebp, esp
        sub esp, 8
        movsd xmm0, QWORD PTR _a$[ebp]
        divsd  xmm0, QWORD PTR __real@40091eb851eb851f
        movsd  xmm1, QWORD PTR _b$[ebp]
        mulsd  xmm1, QWORD PTR __real@4010666666666666
        addsd  xmm0, xmm1
        movsd  QWORD PTR tv70[ebp], xmm0
        fld    QWORD PTR tv70[ebp]
        mov    esp, ebp
        pop    ebp
        ret    0
    _f
    ENDP

```

Listing 1.393: 最適化 MSVC 2012 x86

```

tv67 = 8 ; size = 8
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f
    PROC

```

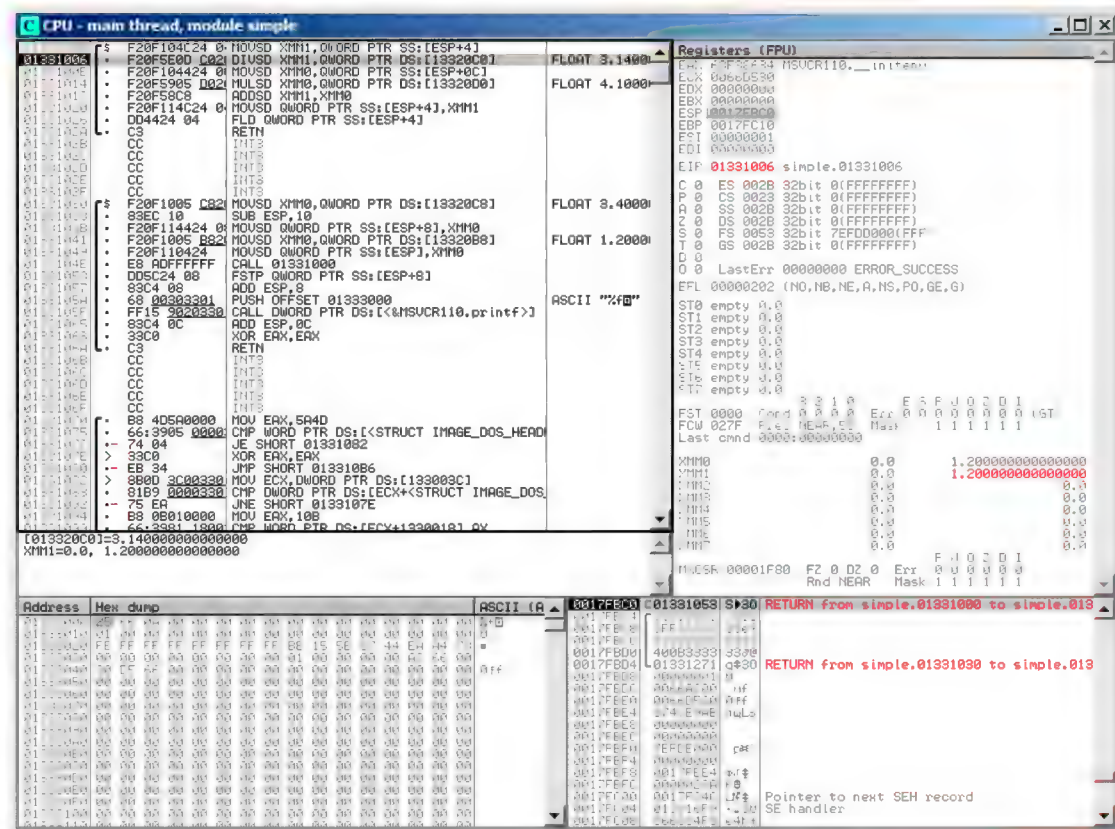
```

movsd    xmm1, QWORD PTR _a$(esp-4)
divsd    xmm1, QWORD PTR __real@40091eb851eb851f
movsd    xmm0, QWORD PTR _b$(esp-4)
mulsd    xmm0, QWORD PTR __real@4010666666666666
addsd    xmm1, xmm0
movsd    QWORD PTR tv67(esp-4), xmm1
fld      QWORD PTR tv67(esp-4)
ret      0
_f       ENDP

```

これはほぼ同じコードですが、呼び出し規約に関していくつかの違いがあります。1) 引数はXMMレジスタではなくスタックに渡されます（FPUの例（[1.19 on page 268](#)）のよう）。2) 関数の結果が ST(0) に返されます。そのためには、XMMレジスタの1つから ST(0) に（ローカル変数 tv を介して）コピーします。

最適化された例を OllyDbg で試してみましょう



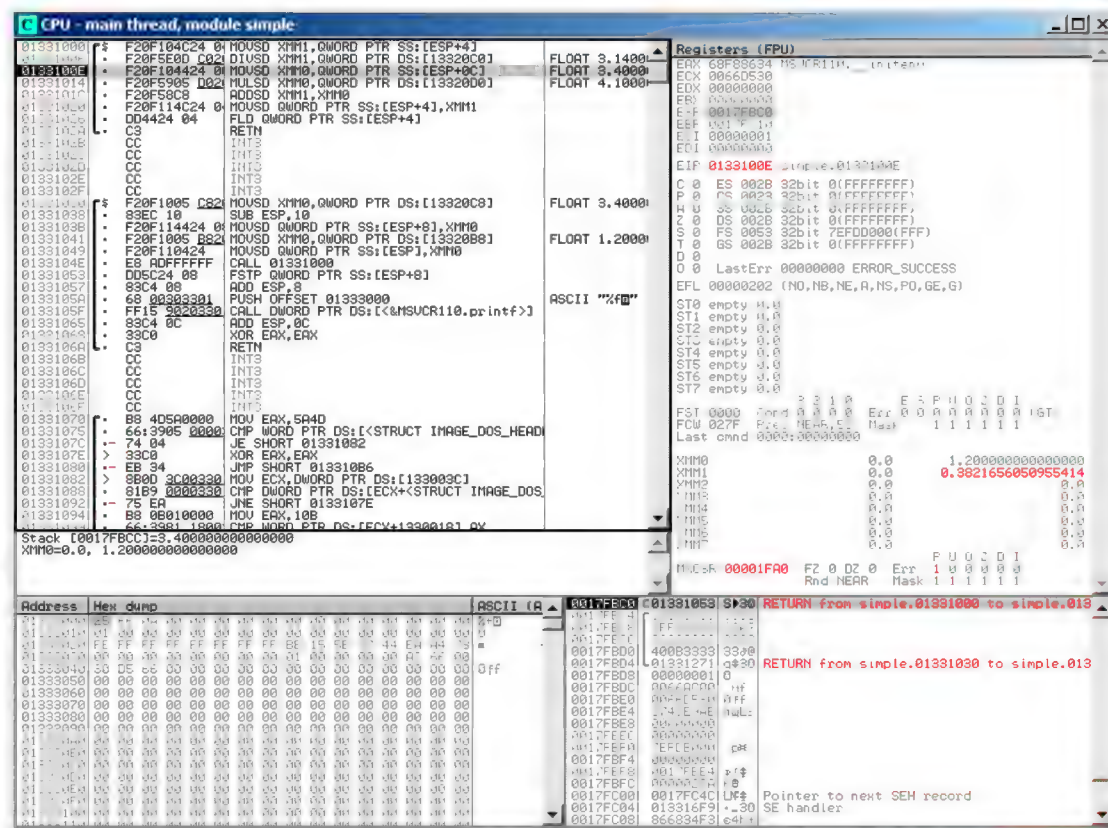


図 1.114: OllyDbq: DIVSD 商を計算し、結果を XMM1 に保存する

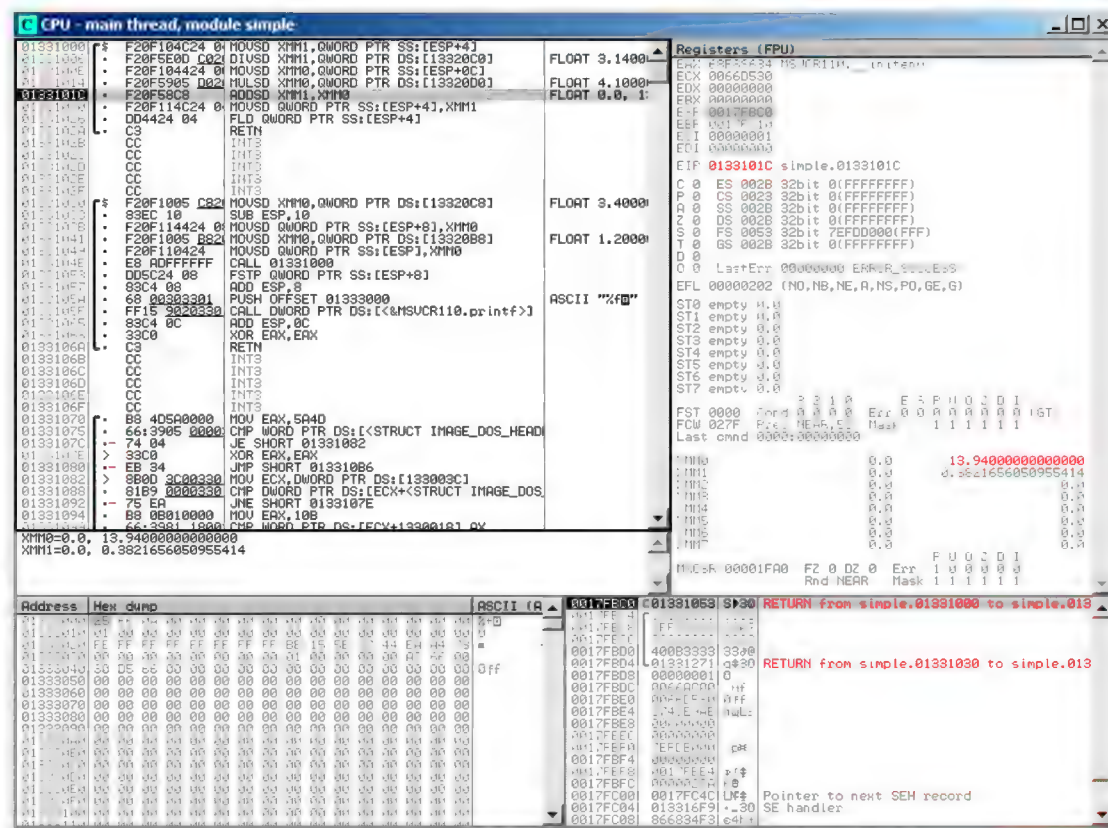


図 1.115: OllyDbg: MULSD calculated 積を計算し、XMM0 に保存する

CPU - main thread, module simple

Address	Hex dump	ASCII (A)	Registers (FPU)
01331000	F20F104C24 0		EAX 68F88C34 MSUCR110, __initenv
01331006	F20F5E00 002		ECX 00F7C500
0133100E	F20F104424 0		EDX 00000000
01331014	F20F5905 002		ESI 00000000
0133101A	F20F5905 002		EDI 0017F6C0
01331020	F20F5905 002		EIP 01331020 simple.01331020
01331026	F20F114424 0		C 0 ES 002B 32bit 0(FFFFFFFF)
0133102C	004424 04		P 0 CS 0023 32bit 0(FFFFFFFF)
01331030	C9		A 0 SS 002B 32bit 0(FFFFFFFF)
01331036	C9		C 0 DS 002B 32bit 0(FFFFFFFF)
0133103C	C9		S 0 FS 0053 32bit 7EFD0000(FFF)
01331040	C9		T 0 GS 003B 32bit 0(FFFFFFFF)
01331046	C9		0 0 LastErr 00000000 ERROR_SUCCESS
0133104C	C9		EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
01331050	C9		ST0 empty 0.0
01331056	C9		ST1 empty 0.0
0133105C	C9		ST2 empty 0.0
01331060	C9		ST3 empty 0.0
01331066	C9		ST4 empty 0.0
0133106C	C9		ST5 empty 0.0
01331070	C9		ST6 empty 0.0
01331076	C9		ST7 empty 0.0
0133107C	C9		ESP 00000000 Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
01331080	C9		FCW 027F Prec NEAR, S3 Mask 1 1 1 1 1 1
01331086	C9		Last cmd 0000:00000000
0133108C	C9		MM0 0.0 13.940000000000000
01331090	C9		MM1 0.0 14.32216560509554
01331096	C9		MM2 0.0 0.0
0133109C	C9		MM3 0.0 0.0
013310A0	C9		MM4 0.0 0.0
013310A6	C9		MM5 0.0 0.0
013310AC	C9		MM6 0.0 0.0
013310B0	C9		MM7 0.0 0.0
013310B6	C9		MXCSR 00001FA0 FZ 0 DZ 0 Err 1 0 0 0 0 0
013310BC	C9		Rnd NEAR Mask 1 1 1 1 1 1
013310C0	C9		
013310C6	C9		
013310CC	C9		
013310D0	C9		
013310D6	C9		
013310DC	C9		
013310E0	C9		
013310E6	C9		
013310EC	C9		
013310F0	C9		
013310F6	C9		
013310FC	C9		
01331100	C9		
01331106	C9		
0133110C	C9		
01331110	C9		
01331116	C9		
0133111C	C9		
01331120	C9		
01331126	C9		
0133112C	C9		
01331130	C9		
01331136	C9		
0133113C	C9		
01331140	C9		
01331146	C9		
0133114C	C9		
01331150	C9		
01331156	C9		
0133115C	C9		
01331160	C9		
01331166	C9		
0133116C	C9		
01331170	C9		
01331176	C9		
0133117C	C9		
01331180	C9		
01331186	C9		
0133118C	C9		
01331190	C9		
01331196	C9		
0133119C	C9		
013311A0	C9		
013311A6	C9		
013311AC	C9		
013311B0	C9		
013311B6	C9		
013311BC	C9		
013311C0	C9		
013311C6	C9		
013311CC	C9		
013311D0	C9		
013311D6	C9		
013311DC	C9		
013311E0	C9		
013311E6	C9		
013311EC	C9		
013311F0	C9		
013311F6	C9		
013311FC	C9		
01331200	C9		
01331206	C9		
0133120C	C9		
01331210	C9		
01331216	C9		
0133121C	C9		
01331220	C9		
01331226	C9		
0133122C	C9		
01331230	C9		
01331236	C9		
0133123C	C9		
01331240	C9		
01331246	C9		
0133124C	C9		
01331250	C9		
01331256	C9		
0133125C	C9		
01331260	C9		
01331266	C9		
0133126C	C9		
01331270	C9		
01331276	C9		
0133127C	C9		
01331280	C9		
01331286	C9		
0133128C	C9		
01331290	C9		
01331296	C9		
0133129C	C9		
013312A0	C9		
013312A6	C9		
013312AC	C9		
013312B0	C9		
013312B6	C9		
013312BC	C9		
013312C0	C9		
013312C6	C9		
013312CC	C9		
013312D0	C9		
013312D6	C9		
013312DC	C9		
013312E0	C9		
013312E6	C9		
013312EC	C9		
013312F0	C9		
013312F6	C9		
013312FC	C9		
01331300	C9		
01331306	C9		
0133130C	C9		
01331310	C9		
01331316	C9		
0133131C	C9		
01331320	C9		
01331326	C9		
0133132C	C9		
01331330	C9		
01331336	C9		
0133133C	C9		
01331340	C9		
01331346	C9		
0133134C	C9		
01331350	C9		
01331356	C9		
0133135C	C9		
01331360	C9		
01331366	C9		
0133136C	C9		
01331370	C9		
01331376	C9		
0133137C	C9		
01331380	C9		
01331386	C9		
0133138C	C9		
01331390	C9		
01331396	C9		
0133139C	C9		
013313A0	C9		
013313A6	C9		
013313AC	C9		
013313B0	C9		
013313B6	C9		
013313BC	C9		
013313C0	C9		
013313C6	C9		
013313CC	C9		
013313D0	C9		
013313D6	C9		
013313DC	C9		
013313E0	C9		
013313E6	C9		
013313EC	C9		
013313F0	C9		
013313F6	C9		
013313FC	C9		
01331400	C9		
01331406	C9		
0133140C	C9		
01331410	C9		
01331416	C9		
0133141C	C9		
01331420	C9		
01331426	C9		
0133142C	C9		
01331430	C9		
01331436	C9		
0133143C	C9		
01331440	C9		
01331446	C9		
0133144C	C9		
01331450	C9		
01331456	C9		
0133145C	C9		
01331460	C9		
01331466	C9		
0133146C	C9		
01331470	C9		
01331476	C9		
0133147C	C9		
01331480	C9		
01331486	C9		
0133148C	C9		
01331490	C9		
01331496	C9		
0133149C	C9		
013314A0	C9		
013314A6	C9		
013314AC	C9		
013314B0	C9		
013314B6	C9		
013314BC	C9		
013314C0	C9		
013314C6	C9		
013314CC	C9		
013314D0	C9		
013314D6	C9		
013314DC	C9		
013314E0	C9		
013314E6	C9		
013314EC	C9		
013314F0	C9		
013314F6	C9		
013314FC	C9		
01331500	C9		
01331506	C9		
0133150C	C9		
01331510	C9		
01331516	C9		
0133151C	C9		
01331520	C9		
01331526	C9		
0133152C	C9		
01331530	C9		
01331536	C9		
0133153C	C9		
01331540	C9		
01331546	C9		
0133154C	C9		
01331550	C9		
01331556	C9		
0133155C	C9		
01331560	C9		
01331566	C9		
0133156C	C9		
01331570	C9		
01331576	C9		
0133157C	C9		
01331580	C9		
01331586	C9		
0133158C	C9		
01331590	C9		
01331596	C9		
0133159C	C9		
013315A0	C9		
013315A6	C9		
013315AC	C9		
013315B0	C9		
013315B6	C9		
013315BC	C9		
013315C0	C9		
013315C6	C9		
013315CC	C9		
013315D0	C9		
013315D6	C9		
013315DC	C9		
013315E0	C9		
013315E6	C9		
013315EC	C9		
013315F0	C9		
013315F6	C9		
013315FC	C9		
01331600	C9		
01331606	C9		
0133160C	C9		
01331610	C9		
01331616	C9		
0133161C	C9		
01331620	C9		
01331626	C9		
0133162C	C9		
01331630	C9		
01331636	C9		
0133163C	C9		
01331640	C9		
01331646	C9		
0133164C	C9		
01331650	C9		
01331656	C9		
0133165C	C9		
01331660	C9		
01331666	C9		

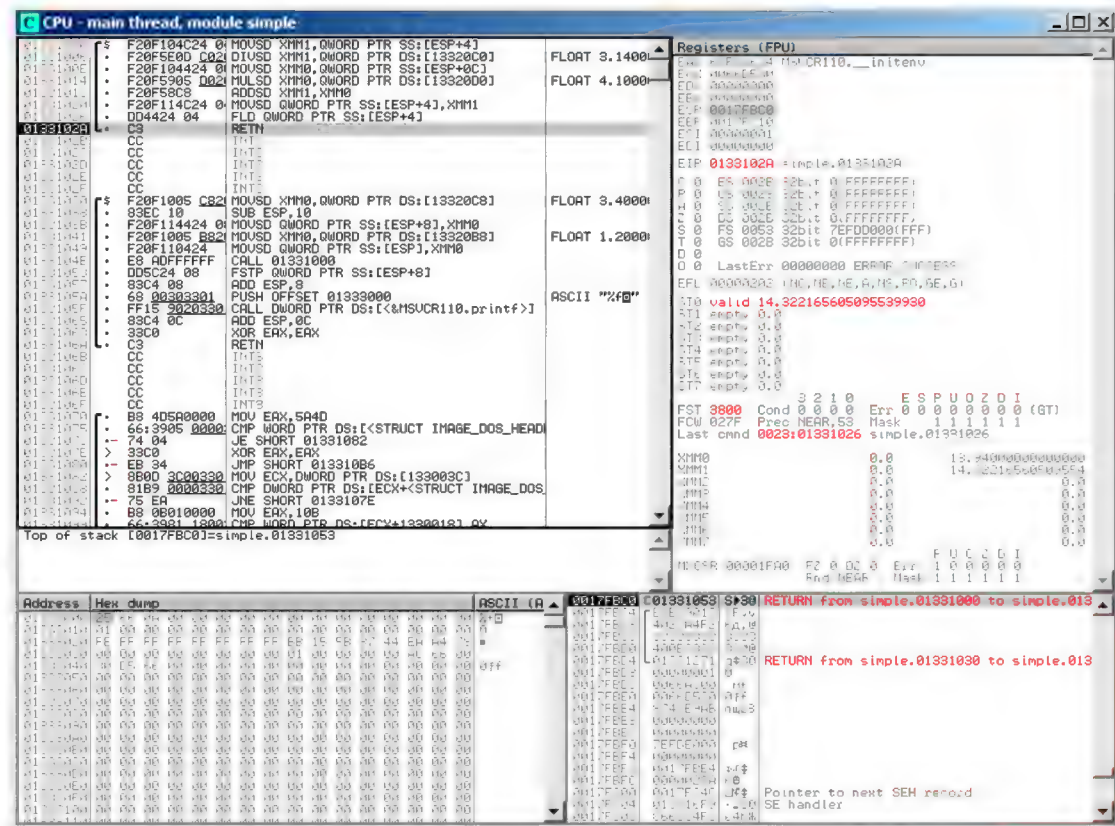


図 1.117: OllyDbg: FLD は関数の結果を ST(0)に残す

OllyDbg はXMMレジスタを *double* 型の数のペアとして示していますが、使用されているのはその低位の部分だけです。

SSE2命令(接尾辞-SD)が現在実行されているため、明らかに、OllyDbg はそれらをその形式で表示します。

しかし、もちろん、レジスタフォーマットを切り替えて、その内容を4つの *float* 数またはちょうど16バイトとして表示することは可能です。

第1.29.2節 引数を介して浮動小数点数を渡す

```
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

    return 0;
}
```

それらは XMM0-XMM3 レジスタの下位半分に渡されます。

Listing 1.394: 最適化 MSVC 2012 x64

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r  ; 1.54

main      PROC
          sub     rsp, 40                      ; 00000028H
          movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
          movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
          call    pow
          lea     rcx, OFFSET FLAT:$SG1354
          movaps  xmm1, xmm0
          movd    rdx, xmm1
          call    printf
          xor     eax, eax
          add     rsp, 40                      ; 00000028H
          ret     0
main      ENDP
```

IntelおよびAMDのマニュアルには MOVSDX 命令がなく (8.1.4 on page 564)、単に MOVSD と呼ばれています。そのため、x86では同じ名前を共有する2つの命令があります (他のものについては ?? on page ?? を参照)。どうやら、Microsoftの開発者たちはこの混乱を取り除きたかったので、MOVSDX に改名しました。XMMレジスタの下半分に値をロードするだけです。

pow() は XMM0 と XMM1 から引数を取り、結果を XMM0 に返します。その後 printf() のために RDX に移動されます。なぜでしょうか？おそらく printf が可変引数関数だからでしょう。

Listing 1.395: 最適化 GCC 4.4.6 x64

```
.LC2:
    .string "32.01 ^ 1.54 = %lf\n"
main:
    sub     rsp, 8
    movsd   xmm1, QWORD PTR .LC0[rip]
    movsd   xmm0, QWORD PTR .LC1[rip]
    call    pow
```

```

; 結果がXMM0にある
mov     edi, OFFSET FLAT:.LC2
mov     eax, 1 ; ベクトルレジスタの数を渡す
call    printf
xor     eax, eax
add     rsp, 8
ret
.LC0:
        .long    171798692
        .long    1073259479
.LC1:
        .long    2920577761
        .long    1077936455

```

GCCはより明確な出力を生成します。printf() の値は XMM0 に渡されます。ところで、printf() のために EAX に1が書かれている場合は、標準で要求されているように [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)]¹⁷⁴、1つの引数がベクトルレジスタに渡されることを意味します。

第1.29.3節 Comparison example

```

#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;

    return b;
};

int main()
{
    printf ("%f\n", d_max (1.2, 3.4));
    printf ("%f\n", d_max (5.6, -4));
};

```

x64

Listing 1.396: 最適化 MSVC 2012 x64

```

a$ = 8
b$ = 16
d_max PROC
        comisd    xmm0, xmm1
        ja        SHORT $LN2@d_max
        movaps    xmm0, xmm1

```

¹⁷⁴以下で利用可能 <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

```
$LN2@d_max:
    fatret    0
d_max      ENDP
```

最適化 MSVCはとても理解しやすいコードを生成します。

COMISD は「Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS」です。本質的に、命令が示すそのもののことをします。

非最適化 MSVC はもっと冗長なコードを生成します。しかし、これもそんなに理解するのが難しくありません。

Listing 1.397: MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max  PROC
    movsdx  QWORD PTR [rsp+16], xmm1
    movsdx  QWORD PTR [rsp+8], xmm0
    movsdx  xmm0, QWORD PTR a$[rsp]
    comisd  xmm0, QWORD PTR b$[rsp]
    jbe     SHORT $LN1@d_max
    movsdx  xmm0, QWORD PTR a$[rsp]
    jmp     SHORT $LN2@d_max
$LN1@d_max:
    movsdx  xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
    fatret  0
d_max  ENDP
```

しかしながら、GCC 4.4.6はもっと最適化して MAXSD (「Return Maximum Scalar Double-Precision Floating-Point Value」) 命令を使用します。これは単に最大値を選択します！

Listing 1.398: 最適化 GCC 4.4.6 x64

```
d_max:
    maxsd   xmm0, xmm1
    ret
```


x86

この例をMSVC 2012の最適化オプションをONにしてコンパイルしてみましょう。

Listing 1.399: 最適化 MSVC 2012 x86

```
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
_d_max PROC
    movsd    xmm0, QWORD PTR _a$[esp-4]
    comisd   xmm0, QWORD PTR _b$[esp-4]
    jbe      SHORT $LN1@d_max
    fld      QWORD PTR _a$[esp-4]
    ret      0
$LN1@d_max:
    fld      QWORD PTR _b$[esp-4]
    ret      0
_d_max ENDP
```

ほとんど同じですが、 a と b の値はスタックからとられて、関数の結果は $ST(0)$ に残ります。

OllyDbg でこの例をロードした場合、COMISD 命令が値を比較して CF と PF フラグをどうやってセット／クリアするのかみることができます。

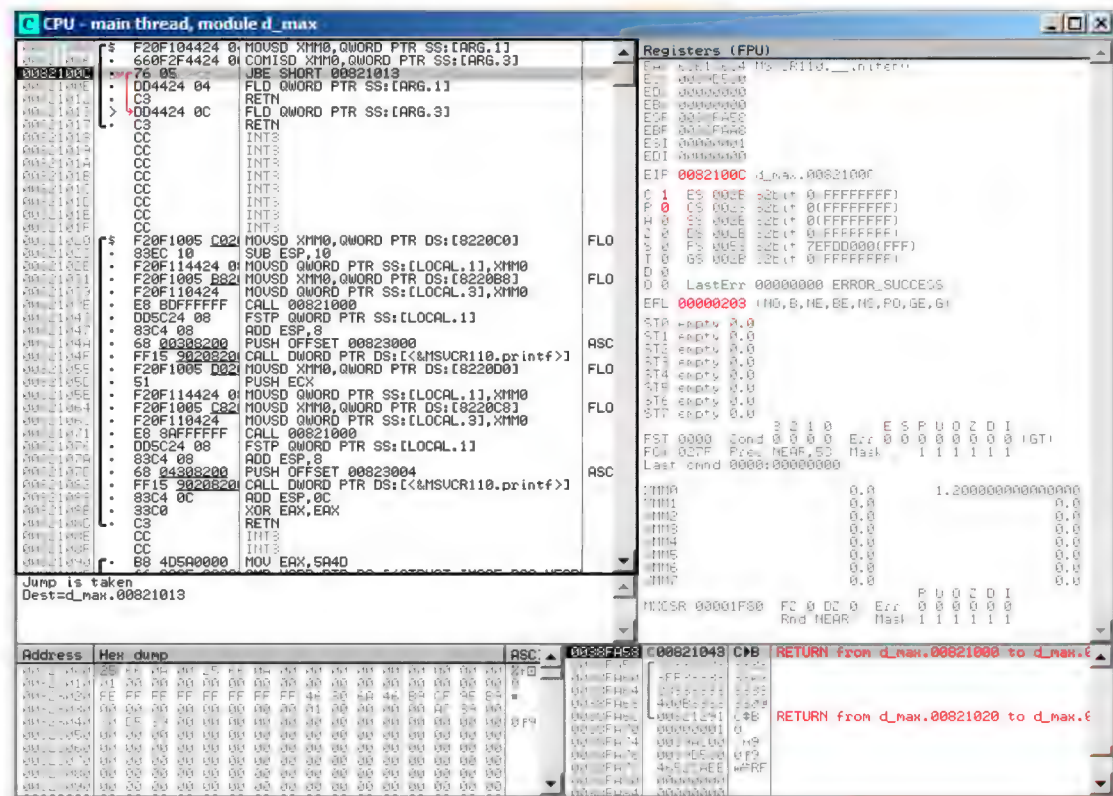


図 1.118: OllyDbg: COMISD は CF と PF フラグを変更

第1.29.4節計算機イプシロンを計算する: x64 と SIMD

「計算機イプシロンを計算する」の例を *double* 型で再訪しましょう: リスト1.24.2

x64でコンパイルします。

Listing 1.400: 最適化 MSVC 2012 x64

```

v$ = 8
calculate_machine_epsilon PROC
    movsdx    QWORD PTR v$[rsp], xmm0
    movaps    xmm1, xmm0
    inc       QWORD PTR v$[rsp]
    movsdx    xmm0, QWORD PTR v$[rsp]
    subsd     xmm0, xmm1
    ret       0
calculate_machine_epsilon ENDP

```

128ビットXMMレジスタに値を1加える方法がないので、メモリ上に配置しなければなりません。

しかしながら、ADDSD 命令 () があります。高位64ビットを無視しつつ、これは低位64ビ

ットのXMMレジスタに値を加えることができます。しかし、MSVC 2012はおそらくそれほど良くないです。¹⁷⁵

それでも、値はXMMレジスタに再ロードされ、減算が行われます。SUBSD は「Subtract Scalar Double-Precision Floating-Point Values」です。つまり、128ビットXMMレジスタの下位64ビット部に対して動作します。結果はXMM0レジスタに返されます。

第1.29.5節疑似乱数値の生成例を再訪

「疑似乱数値の生成例」を再訪しましょう：リスト1.24.1

MSVC 2012でコンパイルすると、FPU用にSIMD命令を使用します。

Listing 1.401: 最適化 MSVC 2012

```
__real@3f800000 DD 03f800000r ; 1

tv128 = -4
_tmp$ = -4
?float_rand@@YAMXZ PROC
    push    ecx
    call    ?my_rand@@YAIXZ
; EAX=疑似乱数値
    and     eax, 8388607 ; 007fffffH
    or      eax, 1065353216 ; 3f800000H
; EAX=疑似乱数値 & 0x007fffff | 0x3f800000
; ローカルスタックに保存
    mov     DWORD PTR _tmp$(esp+4), eax
; 浮動小数点数としてリロード
    movss   xmm0, DWORD PTR _tmp$(esp+4)
; 1.0を減算
    subss   xmm0, DWORD PTR __real@3f800000
; 一時変数に置くことで値をST0にムーブ
    movss   DWORD PTR tv128(esp+4), xmm0
; 値をST0にリロード
    fld     DWORD PTR tv128(esp+4)
    pop     ecx
    ret     0
?float_rand@@YAMXZ ENDP
```

命令はすべて-SS接尾辞がついています。これは、「Scalar Single」を表します。

「Scalar」は、1つの値だけがレジスタに格納されることを意味します。

「Single」¹⁷⁶は *float* データ型を表します。

第1.29.6節概要

ここに示すすべての例では、数値をIEEE 754形式で格納するために、XMMレジスタの下半分だけが使用されています。

¹⁷⁵練習として、ローカルスタックの使用を取り除くためにこのコードを書き直してもいいかもしれません。

¹⁷⁶すなわち、単精度

本質的に、-SD (「Scalar Double-Precision」) の接頭語がついた命令はすべて、IEEE 754形式の浮動小数点数として動作します。そして、XMMレジスタの下位64ビット半分に格納されます。

そしてそれは、おそらくSIMD拡張が過去のFPU拡張よりも混沌としていない方法で進化したために、FPUよりも簡単です。スタックレジスタモデルは使用されません。

double を *float* に置き換えようとした場合

これらの例では、同じ命令が使用されますが、-SS (「Scalar Single-Precision」) 接頭語がつきます。例えば、MOVSS、COMISS、ADDSS などです。

「Scalar」は、SIMDレジスタに複数の値ではなく1つの値しか含まれていないことを意味します。

レジスタ内の複数の値を同時に扱う命令は、それらの名前に「パック」されています。

言うまでもなく、SSE2命令は64ビットのIEEE 754形式の数 (*double*) で機能しますが、FPUの浮動小数点数の内部表現は80ビットの数値です。

したがって、FPUは丸め誤差を少なくすることができ、その結果、FPUはより正確な計算結果を得られます。

第1.30節ARM固有の詳細

第1.30.1節番号の前の番号記号 (#)

Keilコンパイラ、[IDA](#)、およびobjdumpは、すべての番号の前に「#」番号記号を付けます。例えば：リスト[1.16.1](#)。

しかしGCC 4.9がアセンブリ言語出力を生成するとき、それはしません。例えば：リスト[??](#)。

この本のARMのリストは多少複雑です。

どちらの方法が正しいのかわかりにくいです。おそらく、彼/彼女が働く環境で受け入れられている規則に従わなければなりません。

第1.30.2節アドレッシングモード

この命令はARM64でも使用できます。

```
ldr    x0, [x29,24]
```

これは、X29の値に24を加えて、このアドレスから値をロードすることを意味します。

24が括弧の内側にあることに注意してください。数字が括弧の外側にある場合、意味は異なります。

```
ldr    w4, [x1],28
```

これは、X1のアドレスに値をロードしてから、X1に28を加算することを意味します。

ARMでは、ロードに使用されるアドレスに定数を追加したりそこから定数を減算したりできます。

そして、ロードの前後にそれをする事は可能です。

x86にはそのようなアドレッシングモードはありませんが、他のプロセッサには、PDP-11でさえ、存在します。

PDP-11の前置インクリメント、後置インクリメント、前置デクリメント、後置デクリメントの各モードは、(PDP-11上で開発された) そのようなC言語の構造体が `*ptr++`, `++ptr`, `*ptr--`, `--ptr` として出現したことに対して「罪」がありました。

ところで、これはCの機能を暗記するのが難しいことの1つです。このようになっています。

C term	ARM term	C statement	how it works
後置インクリメント	後置インデックスアドレッシング	<code>*ptr++</code>	<code>*ptr</code> の値を使い、次に <code>ptr</code> ポインタをインクリメント
後置デクリメント	後置インデックスアドレッシング	<code>*ptr--</code>	<code>*ptr</code> の値を使い、次に <code>ptr</code> ポインタをデクリメント
前置インクリメント	前置インデックスアドレッシング	<code>++ptr</code>	<code>ptr</code> ポインタをインクリメント、次に <code>*ptr</code> の値を使用
前置デクリメント	前置インデックスアドレッシング	<code>--ptr</code>	<code>ptr</code> の値をデクリメント、次に <code>*ptr</code> の値を使用

ARMのアセンブリ言語では、プレインデクシングに感嘆符が付けられています。たとえば、リスト1.28の2行目を参照してください。

デニス・リッチー（C言語の作成者の一人）は、このプロセッサの機能がPDP-7¹⁷⁷に存在していたため、ケン・トンプソン（もう一人のC言語作成者）によって発明されたと考えていると述べました [Dennis M. Ritchie, *The development of the C language*, (1993)]¹⁷⁸

したがって、C言語コンパイラは、それがターゲットプロセッサに存在する場合は使用できます。

配列処理にはとても便利です。

第1.30.3節レジスタへの定数のロード

32ビットARM

すでにご存じのとおり、すべての命令の長さはARMモードでは4バイト、Thumbモードでは2バイトです。

それでは、1つの命令でエンコードできない場合、どうすれば32ビット値をレジスタにロードできるでしょうか。

やってみましょう。

¹⁷⁷http://yurichev.com/mirrors/C/c_dmr_postincrement.txt

¹⁷⁸以下で利用可能 [pdf](#)

```
unsigned int f()
{
    return 0x12345678;
};
```

Listing 1.402: GCC 4.6.3 -O3 ARMモード

```
f:
    ldr    r0, .L2
    bx     lr
.L2:
    .word  305419896 ; 0x12345678
```

したがって、0x12345678 の値はメモリに保存され、必要に応じてロードされます。
しかし、追加のメモリアクセスを取り除くことは可能です。

Listing 1.403: GCC 4.6.3 -O3 -march=armv7-a (ARMモード)

```
movw    r0, #22136      ; 0x5678
movt     r0, #4660       ; 0x1234
bx       lr
```

値は部分的にレジスタにロードされ、最初に下位部分 (MOVW を使用)、次に上位部分 (MOVT を使用) の順にロードされます。

これは、32ビット値をレジスタにロードするためにARMモードでは2命令が必要であることを意味します。

実際のコードには定数がそれほど多くないため (0と1を除く)、実際のところ問題にはなりません。

2命令のバージョンは1命令のバージョンより遅いということでしょうか？

間違いなくそうです。ほとんどの場合、最新のARMプロセッサはそのようなシーケンスを検出して高速に実行できます。

一方、[IDA](#) はコード内のそのようなパターンを検出し、この関数を次のように逆アセンブルします。

```
MOV     R0, 0x12345678
BX      LR
```

ARM64

```
uint64_t f()
{
    return 0x12345678ABCDEF01;
};
```

Listing 1.404: GCC 4.9.1 -O3

```
mov     x0, 61185      ; 0xef01
```

```
movk    x0, 0xabcd, lsl 16
movk    x0, 0x5678, lsl 32
movk    x0, 0x1234, lsl 48
ret
```

MOVK は「MOV Keep」を表し、つまり、残りのビットには触れずに、レジスタに16ビット値を書き込みます。LSL 接尾辞は、各ステップで値を16、32、48ビット左にシフトします。シフトはロード前に行われます。

これは、64ビット値をレジスタにロードするために4つの命令が必要であることを意味します。

浮動小数点数をレジスタに格納

命令を1つだけ使用して浮動小数点数をDレジスタに格納することは可能です。

例えば：

```
double a()
{
    return 1.5;
};
```

Listing 1.405: GCC 4.9.1 -O3 + objdump

```
0000000000000000 <a>:
0: 1e6f1000          fmov    d0, #1.5000000000000000e+000
4: d65f03c0          ret
```

1.5 という数は実際、32ビット命令でエンコードされます。しかし、どうやって？

ARM64では、いくつかの浮動小数点数をエンコードするために8ビットが FMOV 命令にはあります。

アルゴリズムは [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]¹⁷⁹では VFPEExpandImm() と呼ばれます。minifloat¹⁸⁰とも呼ばれます。

さまざまな値を試すことができます。コンパイラは 30.0 と 31.0 をエンコードできますが、IEEE 754形式で8バイトをこの番号に割り当てる必要があるため、32.0 をエンコードすることはできません。

```
double a()
{
    return 32;
};
```

Listing 1.406: GCC 4.9.1 -O3

```
a:
```

¹⁷⁹以下で利用可能 [http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_\(Issue_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

¹⁸⁰wikipedia


```

        ldr     d0, .LC0
        ret
.LC0:
        .word   0
        .word   1077936128

```

第1.30.4節ARM64での再配置

ご存じのとおり、ARM64には4バイトの命令があるため、単一の命令を使用して大きな数をレジスタに書き込むことは不可能です。

それにもかかわらず、実行可能イメージはメモリ内の任意のランダムアドレスに読み込むことができるので、relocsが存在するのはそのためです。(Win32 PEに関して) relocsについてもっと読む: ?? on page ??

アドレスはARM64の ADRP と ADD 命令のペアを使用して形成されます。

1つ目は4KiBページのアドレスをロードし、2つ目は残りを追加します。win32でGCC (Linaro) 4.9の「ハローワールド!」(リスト1.8) から例をコンパイルしましょう:

Listing 1.407: GCC (Linaro) 4.9 and objdump of object file

```

...>aarch64-linux-gnu-gcc.exe hw.c -c
...>aarch64-linux-gnu-objdump.exe -d hw.o
...
0000000000000000 <main>:
   0:  a9bf7bfd      stp     x29, x30, [sp,#-16]!
   4:  910003fd      mov     x29, sp
   8:  90000000      adrp    x0, 0 <main>
  c:  91000000      add     x0, x0, #0x0
 10:  94000000      bl      0 <printf>
 14:  52800000      mov     w0, #0x0 // #0
 18:  a8c17bfd      ldp     x29, x30, [sp],#16
 1c:  d65f03c0      ret

...>aarch64-linux-gnu-objdump.exe -r hw.o
...
RELOCATION RECORDS FOR [.text]:
OFFSET                TYPE                VALUE
0000000000000008  R_AARCH64_ADR_PREL_PG_HI21  .rodata
000000000000000c  R_AARCH64_ADD_ABS_L012_NC   .rodata
0000000000000010  R_AARCH64_CALL26            printf

```

そのため、このオブジェクトファイルには3つの再配置 (relocs) があります。

- 最初のページアドレスはページアドレスを取り、最下位の12ビットを切り捨て、残りの上位21ビットを ADRP 命令のビットフィールドに書き込みます。これは、下位12ビ

ットをエンコードする必要がないため、ADRP 命令には21ビットのスペースしかありません。

- 2番目のものは、ページ開始に関連するアドレスの12ビットを ADD 命令のビットフィールドに入れます。
- 最後の26ビットの1は、printf() 関数へのジャンプがあるアドレス 0x10 の命令に適用されます。

すべてのARM64（およびARMモードのARM）命令アドレスは、下位2ビットにゼロがあるため（すべての命令のサイズが4バイトであるため）、28ビットアドレス空間の最大26ビット（±128MB）のみをエンコードする必要があります。

実行可能ファイルにはそのような再配置はありません。なぜなら、「Hello!」の文字列がどこにあるか、どのページにあるか、puts() のアドレスもわかっているからです。

そのため、ADRP、ADD、および BL 命令にはすでに値が設定されています（リンクはリンク中にそれらを書き込みました）。

Listing 1.408: objdump of executable file

```
0000000000400590 <main>:
400590:    a9bf7bfd      stp     x29, x30, [sp,#-16]!
400594:    910003fd      mov     x29, sp
400598:    90000000      adrp    x0, 400000 <_init-0x3b8>
40059c:    91192000      add     x0, x0, #0x648
4005a0:    97ffffa0      bl      400420 <puts@plt>
4005a4:    52800000      mov     w0, #0x0 // #0
4005a8:    a8c17bfd      ldp     x29, x30, [sp],#16
4005ac:    d65f03c0      ret

...

Contents of section .rodata:
400640 01000200 00000000 48656c6c 6f210000 .....Hello!...
```

例として、BL命令を手動で逆アセンブルしてみましょう。

0x97ffffa0 は 0b1001011111111111111111111111111110100000 です。[*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)C5.6.26] によると、imm26 は最後の26ビットです。

imm26 = 0b1111111111111111111111111111111101000000 それは 0x3FFFFFFA0 ですが、MSBは1です。従って数は負数です。そして私達のために便利な形式に手動で変換できます。否定の規則に従って、すべてのビットを反転します (0b10111111=0x5F)。そして、1を加算します (0x5F+1=0x60)。そのため、符号付き形式の数は-0x60 です。-0x60 に4を掛けてみましょう。(オペコードに格納されているアドレスは4で除算されているため) -0x180 になります。それでは、宛先アドレスを計算しましょう: 0x4005a0 + (-0x180) = 0x400420 (注意: 現在のPC!の値ではなく、BL命令のアドレスを考慮します。異なるかもしれません!) そのため、宛先アドレスは 0x400420 です。

ARM64関連の再配置の詳細: [*ELF for the ARM 64-bit Architecture (AArch64)*, (2013)]¹⁸¹

¹⁸¹以下で利用可能 http://infocenter.arm.com/help/topic/com.arm.doc.ih10056b/IHI0056B_aaelf64.pdf

第1.31節MIPS特有の詳細

第1.31.1節32ビット定数をレジスタにロードする

```
unsigned int f()
{
    return 0x12345678;
};
```

MIPSのすべての命令は、ARMと同様に32ビットのサイズを持っているため、1つの命令に32ビットの定数を埋め込むことはできません。

そのため、少なくとも2つの命令を使用する必要があります。1つ目は32ビット数の上位部分をロードし、2つ目はターゲットレジスタの下位16ビット部分を効果的に設定するOR演算を適用します。

Listing 1.409: GCC 4.4.5 -O3 (アセンブリ出力)

```
li    $2,305397760 # 0x12340000
j     $31
ori   $2,$2,0x5678 ; 分岐遅延スロット
```

IDA はこのような頻繁に発生するコードパターンを完全に認識しているので、便宜上、最後の ORI 命令を LI 疑似命令として示しています。それは完全な32ビット数を \$V0 レジスタにロードするとされています。

Listing 1.410: GCC 4.4.5 -O3 (IDA)

```
lui    $v0, 0x1234
jr     $ra
li     $v0, 0x12345678 ; 分岐遅延スロット
```

GCCアセンブリの出力にはLI疑似命令がありますが、実際には LUI (「Load Upper Immediate」) であり、これが16ビット値をレジスタの上位部分に格納します。

objdump の出力を見てみましょう。

Listing 1.411: *objdump*

```
00000000 <f>:
0:  3c021234      lui    v0,0x1234
4:  03e00008      jr     ra
8:  34425678      ori    v0,v0,0x5678
```

32ビットグローバル変数をレジスタにロードする

```
unsigned int global_var=0x12345678;

unsigned int f2()
{
    return global_var;
};
```

これは少し異なります。LUI は *global_var* から上位16ビットを \$2（または \$V0）にロードし、次に下位16ビットをロードして \$2 の内容を合計します。

Listing 1.412: GCC 4.4.5 -O3 (アセンブリ出力)

```
f2:
    lui    $2,%hi(global_var)
    lw     $2,%lo(global_var)($2)
    j      $31
    nop    ; 分岐遅延スロット

    ...

global_var:
    .word  305419896
```

IDA はよく使用される LUI/LW 命令ペアを完全に認識しているため、両方を1つの LW 命令に統合します。

Listing 1.413: GCC 4.4.5 -O3 (IDA)

```
_f2:
    lw     $v0, global_var
    jr     $ra
    or     $at, $zero      ; 分岐遅延スロット

    ...

    .data
    .globl global_var
global_var:
    .word  0x12345678      # DATA XREF: _f2
```

objdump の出力はGCCのアセンブリ出力と同じです。オブジェクトファイルの再配置もダンプしましょう。

Listing 1.414: *objdump*

```
objdump -D filename.o

...

0000000c <f2>:
   c:  3c020000      lui     v0,0x0
  10:  8c420000      lw      v0,0(v0)
  14:  03e00008      jr      ra
  18:  00200825      move    at,at      ; 分岐遅延スロット
 1c:  00200825      move    at,at

Disassembly of section .data:

00000000 <global_var>:
   0:  12345678      beq     s1,s4,159e4 <f2+0x159d8>

...
```

```
objdump -r filename.o

...

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE           VALUE
0000000c R_MIPS_HI16         global_var
00000010 R_MIPS_L016         global_var

...
```

global_var のアドレスは、実行可能ファイルのロード中に LUI および LW 命令に直接書き込まれることがわかります。*global_var* の上位16ビット部分は最初のもの（LUI）に、下位16ビット部分は2番目のもの（LW）に入ります。

第1.31.2節MIPSについてさらに読む

Dominic Sweetman, *MIPS Run* 第二版を参照, (2010).

第 2 章

Japanese text placeholder

第2.1節Integral datatypes

整数型とは、数値に変換できる値の型です。これには数値・列挙・ブール値などがあります。

第2.1.1節Bit

ビットの明確な使い方はブール値です。: 0は *false*、1は *true* になります。

ブーリアンのセットはwordにまとめることができます。32ビットのwordには32のブーリアンがあります。これは *bitmap* または *bitfield* と呼ばれています。

しかし、これには明らかなオーバーヘッドがあります。ブール変数にword (または *int*) を使うのは経済的ではありませんが、非常に高速です。

C/C++では0が *false*、0以外の値が *true* になります。例えば:

```
if (1234)
    printf ("this will always be executed\n");
else
    printf ("this will never\n");
```

これはCの文字列を列挙する一般的な方法です:

```
char *input=...;

while(*input) // execute body if *input character is non-zero
{
    // do something with *input
    input++;
};
```

第2.1.2節 Nibble AKA nybble

ハーフバイト、テトラデとして知られています!¹。4ビットに相当します。

これらの用語はいずれも現在でも使用されています。

Binary-coded decimal (BCD²)

4ビットニブルは、有名なインテル4004（電卓で使用された）のような4ビットCPUで使用されていました。

4ビットを使って10進数を表現する *binary-coded decimal (BCD)* という方法があったのは興味深いことです。10進数0は0b0000、10進数9は0b1001で表され、それ以上の値は使用されません。10進数1234は0x1234で表されます。もちろんこの方法は経済的ではありません。

それにもかかわらず、この方法には1つの利点があります。それは10進数からBCDへの変換が容易なことです。BCD は加算、減算などが可能ですが、追加の修正が必要です。x86 CPUにはそのための1Binary-Coded Decimal 443という珍しい命令があります。: AAA/DAA (加算後に調整), AAS/DAS (減算後に調整), AAM (乗算後に調整), AAD (除算後に調整)。

CPUがBCDをサポートする必要があるので、*half-carry flag* (on 8080/Z80) や *auxiliary flag* (AF on x86) が存在します。これは下位4ビットを進めた後に生成されるキャリーフラグで、調整命令に使用されます。

変換が容易なことから [Peter Abel, *IBM PC assembly language and programming* (1987)] という本が人気になりました。この本はさておき、著者は *magic numbers*(?? on page ??) 以外にBCDナンバーを見たことはありません。例えば、誰かの誕生日が0x19791011のようにエンコードされている時、これは確かにBCDナンバーです。

驚くべきことに、著者はSAPソフトウェア上でBCDにエンコードされた数字が使われていることに気がつきました。<https://yurichev.com/blog/SAP/>. 価格を含めたいくつかの数字は、データベース上ではBCD形式でエンコードされています。おそらく、何らかの古いソフトウェア・ハードウェアとの互換性を持たせるために使用されたのではないのでしょうか？

x86のBCD命令は他の目的、特に文書化されていない方法などでよく使われていました。例えば

```
cmp al,10
sbb al,69h
das
```

この曖昧なコードは、0～15の数字をASCII文字'0'..'9', 'A'..'F' に変換します。

Z80

¹として知られています!

²Binary-Coded Decimal

Z80は8ビットのIntel 8080のCPUのクローンで、スペースの制約から4ビットのALUを持っています。この副作用として *half-carry flag* が容易に、かつ自然に発生することがあります。

第2.1.3節Byte

Byte is primarily used for character storage. 8-bit bytes were not common as today. Punched tapes for teletypes had 5 and 6 possible holes, this is 5 or 6 bits for byte.

To emphasize the fact the byte has 8 bits, byte is sometimes called *octet*: at least *fetchmail* uses this terminology.

9-bit bytes used to exist in 36-bit architectures: 4 9-bit bytes would fit in a single [word](#). Probably because of this fact, C/C++ standard tells that *char* has to have a room for *at least* 8 bits, but more bits are allowable.

For example, in the early C language manual³, we can find this:

char	one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
------	--

By H6070 they probably meant Honeywell 6070, with 36-bit words.

Standard ASCII table

7-bit ASCII table is standard, which has only 128 possible characters. Early E-Mail transport software were operating only on 7-bit ASCII codes, so a [MIME](#)⁴ standard needed to encode messages in non-Latin writing systems. 7-bit ASCII code was augmented by parity bit, resulting in 8 bits.

Data Encryption Standard ([DES](#)⁵) has a 56 bits key, this is 8 7-bit bytes, leaving a space to parity bit for each character.

There is no need to memorize whole [ASCII](#) table, but rather ranges. [0..0x1F] are control characters (non-printable). [0x20..0x7E] are printable ones. Codes starting at 0x80 are usually used for non-Latin writing systems and/or pseudographics.

Significant codes which will be easily memorized are: 0 (end of C-string, '\0' in C/C++); 0xA or 10 (*line feed*, '\n' in C/C++); 0xD or 13 (*carriage return*, '\r' in C/C++).

0x20 (space) is also often memorized.

8-bit CPUs

x86 has capability to work with byte(s) on register level (because they are descendants of 8-bit 8080 CPU), RISC CPUs like ARM and MIPS—not.

³<https://yurichev.com/mirrors/C/bwk-tutor.html>

⁴Multipurpose Internet Mail Extensions

⁵Data Encryption Standard

第2.1.4節 Wide char

This is an attempt to support multi-lingual environment by extending byte to 16-bit. Most well-known example is Windows NT kernel and win32 functions with *W* suffix. This is why each Latin character in plain English text string is interleaved with zero byte. This encoding is called UCS-2 or UTF-16

Usually, *wchar_t* is synonym to 16-bit *short* data type.

第2.1.5節 Signed integer vs unsigned

Some may argue, why unsigned data types exist at first place, since any unsigned number can be represented as signed. Yes, but absence of sign bit in a value extends its range twice. Hence, signed byte has range of -128..127, and unsigned one: 0..255. Another benefit of using unsigned data types is self-documenting: you define a variable which can't be assigned to negative values.

Unsigned data types are absent in Java, for which it's criticized. It's hard to implement cryptographical algorithms using boolean operations over signed data types.

Values like 0xFFFFFFFF (-1) are used often, mostly as error codes.

第2.1.6節 Word

Word word is somewhat ambiguous term and usually denotes a data type fitting in **GPR**. Bytes are practical for characters, but impractical for other arithmetical calculations.

Hence, many **CPUs** have **GPRs** with width of 16, 32 or 64 bits. Even 8-bit CPUs like 8080 and Z80 offer to work with 8-bit register pairs, each pair forming a 16-bit *pseudoregister* (*BC*, *DE*, *HL*, etc.). Z80 has some capability to work with register pairs, and this is, in a sense, some kind of 16-bit CPU emulation.

In general, if a CPU marketed as "n-bit CPU", this usually means it has n-bit **GPRs**.

There was a time when hard disks and **RAM** modules were marketed as having *n* kilo-words instead of *b* kilobytes/megabytes.

For example, *Apollo Guidance Computer* has 2048 words of **RAM**. This was a 16-bit computer, so there was 4096 bytes of **RAM**.

TX-0 had 64K of 18-bit words of magnetic core memory, i.e., 64 kilo-words.

DECSYSTEM-2060 could have up to 4096 kilowords of *solid state memory* (i.e., hard disks, tapes, etc). This was 36-bit computer, so this is 18432 kilobytes or 18 megabytes.

Essentially, why do you need bytes if you have words? Mostly for text strings processing. Words can be used in almost any other situations.

int in C/C++ is almost always mapped to **word**. (Except of AMD64 architecture where *int* is still 32-bit one, perhaps, for the reason of better portability.)

int is 16-bit on PDP-11 and old MS-DOS compilers. *int* is 32-bit on VAX, on x86 starting at 80386, etc.

Even more than that, if type declaration for a variable is omitted in C/C++ program, *int* is used silently by default. Perhaps, this is inheritance of B programming language⁶.

GPR is usually fastest container for variable, faster than packed bit, and sometimes even faster than byte (because there is no need to isolate a single bit/byte from **GPR**). Even if you use it as a container for loop counter in 0..99 range.

Word in assembly language is still 16-bit for x86, because it was so for 16-bit 8086. *Double word* is 32-bit, *quad word* is 64-bit. That's why 16-bit words are declared using DW in x86 assembly, 32-bit ones using DD and 64-bit ones using DQ.

Word is 32-bit for ARM, MIPS, etc., 16-bit data types are called *half-word* there. Hence, *double word* on 32-bit RISC is 64-bit data type.

GDB has the following terminology: *halfword* for 16-bit, **word** for 32-bit and *giant word* for 64-bit.

16-bit C/C++ environment on PDP-11 and MS-DOS has *long* data type with width of 32 bits, perhaps, they meant *long word* or *long int*?

32-bit C/C++ environment has *long long* data type with width of 64 bits.

Now you see why the *word* word is ambiguous.

Should I use *int*?

Some people argue that *int* shouldn't be used at all, because its ambiguity can lead to bugs. For example, well-known *lzhu* library uses *int* at one point and everything works fine on 16-bit architecture. But if ported to architecture with 32-bit *int*, it can crash: <http://yurichev.com/blog/lzhuf/>.

Less ambiguous types are defined in *stdint.h* file: *uint8_t*, *uint16_t*, *uint32_t*, *uint64_t*, etc.

Some people like Donald E. Knuth proposed⁷ more sonorous words for these types: *byte*/*wyde*/*tetrabyte*/*octabyte*. But these names are less popular than clear terms with inclusion of *u* (*unsigned*) character and number right into the type name.

Word-oriented computers

Despite the ambiguity of the **word** term, modern computers are still word-oriented: **RAM** and all levels of cache are still organized by words, not by bytes. However, size in bytes is used in marketing.

Access to RAM/cache by address aligned by word boundary is often cheaper than non-aligned.

⁶<http://yurichev.com/blog/typeless/>

⁷<http://www-cs-faculty.stanford.edu/~uno/news98.html>

During data structures development, which are supposed to be fast and efficient, one should always take into consideration length of the [word](#) on the CPU to be executed on. Sometimes the compiler will do this for programmer, sometimes not.

第2.1.7節Address register

For those who fostered on 32-bit and/or 64-bit x86, and/or RISC of 90s like ARM, MIPS, PowerPC, it's natural that address bus has the same width as [GPR](#) or [word](#). Nevertheless, width of address bus can be different on other architectures.

8-bit Z80 can address 2^{16} bytes, using 8-bit registers pairs or dedicated registers (*IX*, *IY*). *SP* and *PC* registers are also 16-bit ones.

Cray-1 supercomputer has 64-bit GPRs, but 24-bit address registers, so it can address 2^{24} (16 megawords or 128 megabytes). RAM was very expensive in 1970s, and a typical Cray had 1048576 (0x100000) words of RAM or 8MB. So why to allocate 64-bit register for address or pointer?

8086/8088 CPUs had a really weird addressing scheme: values of two 16-bit registers were summed in a weird manner resulting in a 20-bit address. Perhaps, this was some kind of toy-level virtualization (?? on page ??)? 8086 could run several programs (not simultaneously, though).

Early ARM1 has an interesting artifact:

Another interesting thing about the register file is the PC register is missing a few bits. Since the ARM1 uses 26-bit addresses, the top 6 bits are not used. Because all instructions are aligned on a 32-bit boundary, the bottom two address bits in the PC are always zero. These 8 bits are not only unused, they are omitted from the chip entirely.

(<http://www.righto.com/2015/12/reverse-engineering-arm1-ancestor-of.html>)

Hence, it's physically not possible to push a value with one of two last bits set into PC register. Nor it's possible to set any bits in high 6 bits of PC.

x86-64 architecture has virtual 64-bit pointers/addresses, but internally, width of address bus is 48 bits (seems enough to address 256TB of [RAM](#)).

第2.1.8節Numbers

What are numbers used for?

When you see some number(s) altering in a CPU register, you may be interested in what this number means. It's an important skill for a reverse engineer to determine possible data type from a set of changing numbers.

Boolean

If the number is switching from 0 to 1 and back, most chances that this value has boolean data type.

Loop counter, array index

Variable increasing from 0, like: 0, 1, 2, 3...—a good chance this is a loop counter and/or array index.

Signed numbers

If you see a variable which holds very low numbers and sometimes very high numbers, like 0, 1, 2, 3, and 0xFFFFFFFF, 0xFFFFFFFFE, 0xFFFFFFFFD, there's a good chance it is a signed variable in *two's complement* form, and last 3 numbers are -1, -2, -3.

32-bit numbers

There are numbers so large, that there is even a special notation which exists to represent them (Knuth's up-arrow notation). These numbers are so large so these are not practical for engineering, science and mathematics.

Almost all engineers and scientists are happy with IEEE 754 double precision floating point, which has maximal value around $1.8 \cdot 10^{308}$. (As a comparison, the number of atoms in the observable universe, is estimated to be between $4 \cdot 10^{79}$ and $4 \cdot 10^{81}$.)

In fact, upper bound in practical computing is much, much lower. In MS-DOS era 16-bit *int* was used almost for everything (array indices, loop counters), while 32-bit *long* was used rarely.

During advent of x86-64, it was decided for *int* to stay as 32 bit size integer, because, probably, usage of 64-bit *int* is even rarer.

I would say, 16-bit numbers in range 0..65535 are probably most used numbers in computing.

Given that, if you see unusually large 32-bit value like 0x87654321, this is a good chance this can be:

- this can still be a 16-bit number, but signed, between 0xFFFF8000 (-32768) and 0xFFFFFFFF (-1).
- address of memory cell (can be checked using memory map feature of debugger).
- packed bytes (can be checked visually).
- bit flags.
- something related to (amateur) cryptography.
- magic number (?? on page ??).
- IEEE 754 floating point number (can also be checked).

Almost same story for 64-bit values.

...so 16-bit *int* is enough for almost everything?

It's interesting to note: in [Michael Abrash, *Graphics Programming Black Book*, 1997 chapter 13] we can find that there are plenty cases in which 16-bit variables are just enough. In a meantime, Michael Abrash has a pity that 80386 and 80486 CPUs has so little available registers, so he offers to put two 16-bit values into one 32-bit register and then to rotate it using ROR reg, 16 (on 80386 and later) (ROL reg, 16 will also work) or BSWAP (on 80486 and later) instruction.

That reminds us Z80 with alternate pack of registers (suffixed with apostrophe), to which CPU can switch (and then switch back) using EXX instruction.

Size of buffer

When a programmer needs to declare the size of some buffer, values in form of 2^x are usually used (512 bytes, 1024, etc.). Values in 2^x form are easily recognizable (1.21.5 on page 390) in decimal, hexadecimal and binary base.

But needless to say, programmers are still humans with their decimal culture. And somehow, in DBMS area, size of textual database fields is often chosen as 10^x number, like 100, 200. They just think 「Okay, 100 is enough, wait, 200 will be better」. And they are right, of course.

Maximum width of VARCHAR2 data type in Oracle RDBMS is 4000 characters, not 4096.

There is nothing wrong with this, this is just a place where numbers like 10^x can be encountered.

Address

It's always a good idea to keep in mind an approximate memory map of the process you currently debug. For example, many win32 executables started at 0x00401000, so an address like 0x00451230 is probably located inside executable section. You'll see addresses like these in the EIP register.

Stack is usually located somewhere below.

Many debuggers are able to show the memory map of the debuggee, for example: 1.9.3 on page 100.

If a value is increasing by step 4 on 32-bit architecture or by step 8 on 64-bit one, this probably sliding address of some elements of array.

It's important to know that win32 doesn't use addresses below 0x10000, so if you see some number below this constant, this cannot be an address (see also: <https://msdn.microsoft.com/en-us/library/ms810627.aspx>).

Anyway, many debuggers can show you if the value in a register can be an address to something. OllyDbg can also show an ASCII string if the value is an address of it.

Bit field

If you see a value where one (or more) bit(s) are flipping from time to time like 0xABCD1234 → 0xABCD1434 and back, this is probably a bit field (or bitmap).

Packed bytes

When *strcmp()* or *memcmp()* copies a buffer, it loads/stores 4 (or 8) bytes simultaneously, so if a string containing 「4321」, and it would be copied to another place, at one point you'll see 0x31323334 value in some register. This is 4 packed bytes into a 32-bit value.

第 3 章

第 4 章

Java

第4.1節Java

第4.1.1節

第4.1.2節

第4.1.3節

第 5 章

第5.1節Linux

第5.2節Windows NT

第5.2.1節Windows SEH

SEH

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]¹, [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]².

¹以下で利用可能 <http://www.microsoft.com/msj/0197/Exception/Exception.aspx>

²以下で利用可能 <http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf>

第 6 章

ツール

Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education. However, for our freedom's sake, we need free (libre) reverse engineering tools to replace the proprietary tools described in this book.

Richard M. Stallman

第6.1節バイナリ解析

プロセスを実行しないときに使用するツール。

- (フリー、オープンソース) *ent*¹: エントロピー分析ツール。エントロピーについての詳細: ?? on page ??.
- *Hiew*²: バイナリファイルでのコードの小さな変更を追う。アセンブラ／ディスアセンブラを内蔵。
- (フリー、オープンソース) *GHex*³: Linux用の単純な16進エディタ
- (フリー、オープンソース) *xxd* and *od*: ダンプするための標準的なUNIXユーティリティ
- (フリー、オープンソース) *strings*: 実行可能ファイルを含むバイナリファイルでASCII文字列を検索するための*NIXツール Sysinternalsには、ワイド文字列をサポートする代替機能⁴ があります (Windowsで広く使用されているUTF-16)
- (フリー、オープンソース) *Binwalk*⁵: ファームウェアイメージの分析

¹<http://www.fourmilab.ch/random/>

²hiew.ru

³<https://wiki.gnome.org/Apps/Ghex>

⁴<https://technet.microsoft.com/en-us/sysinternals/strings>

⁵<http://binwalk.org/>

- (フリー、オープンソース) *binary grep*: 実行不可能なファイルを含む大きなファイルの中のバイトシーケンスを検索するための小さなユーティリティ: [GitHub](#) 同じ目的のためにrada.reにrafind2もあります。

第6.1.1節ディスアセンブラ

- *IDA*. 古いフリーウェアバージョンがダウンロード可能です ⁶. ホットキーチートシート: ?? on page ??
- *Binary Ninja* ⁷
- (フリー、オープンソース) *zynamics BinNavi* ⁸
- (フリー、オープンソース) *objdump*: ダンプとディスアセンブルのための簡単なコマンドラインユーティリティ
- (フリー、オープンソース) *readelf* ⁹: ELFファイルに関するダンプ情報。

第6.1.2節デコンパイラ

公に利用可能な既知の高品質なCコードへのデコンパイラがたった1つだけあります: *Hex-Rays*:

hex-rays.com/products/decompiler/

詳細を読む: ?? on page ??.

第6.1.3節パッチの比較/diffing

実行可能ファイルの一部を元のバージョンと比較し、パッチが適用されたものとその理由を調べるときに使用することをお勧めします。

- (フリー) *zynamics BinDiff* ¹⁰
- (フリー、オープンソース) *Diaphora* ¹¹

第6.2節ライブ解析

稼働中のシステム上またはプロセスの実行中に使用するツール。

第6.2.1節デバグ

- (フリー) *OllyDbg*. とても人気のあるユーザーモードのwin32デバグ ¹². ホットキーチートシート: ?? on page ??

⁶hex-rays.com/products/ida/support/download_freeware.shtml

⁷<http://binary.ninja/>

⁸<https://www.zynamics.com/binnavi.html>

⁹<https://sourceware.org/binutils/docs/binutils/readelf.html>

¹⁰<https://www.zynamics.com/software.html>

¹¹<https://github.com/joxeankoret/diaphora>

¹²ollydbg.de


```
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) ↗
↳ = 0xb75b3000
```

Mac OS X は同じことを行うために `dtruss`があります。

Cygwinには`strace`もありますが、知る限り、cygwin環境用にコンパイルされた.exeファイルに対してのみ動作します。

第6.2.4節 ネットワーク傍受

Sniffing は興味のある情報を傍受します。

(フリー、オープンソース) *Wireshark*²⁰ ネットワーク傍受のために。また、USBスニフリング機能も備えています。²¹

Wiresharkには若い（または古い）兄弟がいます：*tcpdump*²²、簡単なコマンドラインツールです。

第6.2.5節 Sysinternals

(フリー) Sysinternals (Mark Russinovichによって開発)²³。少なくともこれらツールは重要で、検討する価値があります：プロセスエクスプローラ、Handle、VMMap、TCPView、プロセスモニタ

第6.2.6節 Valgrind

(フリー、オープンソース) メモリリークを検出する強力なツール：<http://valgrind.org/>。強力なJIT²⁴メカニズムのため、Valgrindは他のツールのフレームワークとして使用されています

第6.2.7節 エミュレータ

- (フリー、オープンソース) *QEMU*²⁵：さまざまなCPUおよびアーキテクチャ用のエミュレータ
- (フリー、オープンソース) *DosBox*²⁶：MS-DOSエミュレータ、主にレトロゲームに使用されます。
- (フリー、オープンソース) *SimH*²⁷：大昔のコンピュータ、メインフレームなどのエミュレータ

²⁰<https://www.wireshark.org/>

²¹<https://wiki.wireshark.org/CaptureSetup/USB>

²²<http://www.tcpdump.org/>

²³<https://technet.microsoft.com/en-us/sysinternals/bb842062>

²⁴Just-In-Time compilation

²⁵<http://qemu.org>

²⁶<https://www.dosbox.com/>

²⁷<http://simh.trailing-edge.com/>

第6.3節他のツール

Microsoft Visual Studio Express ²⁸: 簡単な実験に便利な、Visual Studioの無償版。

いくつかの便利なオプション: ?? on page ??.

“Compiler Explorer” というWebサイトがあり、小さなコードスニペットをコンパイルし、さまざまなGCCのバージョンとアーキテクチャ（少なくともx86、ARM、MIPS）で出力を見ることができます：<http://godbolt.org/>—もし私がそれについて知っていたら、私は本のためにそれを使ったでしょう！

第6.3.1節電卓

リバースエンジニアのニーズに合った良い電卓は、少なくとも10進数、16進数、2進数ベース、XORやシフトなどの多くの重要な演算をサポートする必要があります。

- IDA にはビルトインの電卓があります (“?”).
- rada.re には *rax2* があります。
- <https://yurichev.com/progcalc/>
- 最後の手段として、Windowsの標準電卓にはプログラマモードがあります。

第6.4節何か足りないものは？

ここにリストされていない素晴らしいツールが知っている場合は：
[my emails](#)

第6.5節

第6.6節

[Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.](#)

[How to Hack an Expensive Camera and Not Get Killed by Your Wife.](#)

²⁸visualstudio.com/en-US/products/visual-studio-express-vs

第 7 章

その他

第 8 章

読むべき本/ブログ

第8.1節本と他の資料

第8.1.1節リバースエンジニアリング

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)
- Reginald Wong, *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*, (2018)

そして、Kris Kasperskyの本も。

第8.1.2節Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The “Ultimate” Anti-Debugging Reference¹

:

- [Microsoft: Raymond Chen](#)
- [nynaeve.net](#)

¹<http://pferrie.host22.com/papers/antidebug.pdf>

第8.1.3節C/C++

- Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, 2ed, (1988)
- *ISO/IEC 9899:TC3 (C C99 standard)*, (2007)²
- Bjarne Stroustrup, *The C++ Programming Language*, 4th Edition, (2013)
- C++11 standard³
- Agner Fog, *Optimizing software in C++* (2015)⁴
- Marshall Cline, *C++ FAQ*⁵
- Dennis Yurichev, *C/C++ programming language notes*⁶
- JPL Institutional Coding Standard for the C Programming Language⁷

第8.1.4節x86 / x86-64

- Intelマニュアル⁸
- AMDマニュアル⁹
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)¹⁰
- Agner Fog, *Calling conventions* (2015)¹¹
- *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)
- *Software Optimization Guide for AMD Family 16h Processors*, (2013)

やや時代遅れですが、それでも興味深く読めます。

Michael Abrash, *Graphics Programming Black Book*, 1997¹² (彼は、Windows NT 3.1やid Quakeなどのプロジェクトのための低レベルの最適化に関する仕事で知られています。)

第8.1.5節ARM

- ARMマニュアル¹³
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)

²以下で利用可能 <http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf>

³以下で利用可能 <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>.

⁴以下で利用可能 http://agner.org/optimize/optimizing_cpp.pdf.

⁵以下で利用可能 <http://www.parashift.com/c++-faq-lite/index.html>

⁶以下で利用可能 <http://yurichev.com/C-book.html>

⁷以下で利用可能 https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf

⁸以下で利用可能 <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

⁹以下で利用可能 <http://developer.amd.com/resources/developer-guides-manuals/>

¹⁰以下で利用可能 <http://agner.org/optimize/microarchitecture.pdf>

¹¹以下で利用可能 http://www.agner.org/optimize/calling_conventions.pdf

¹²以下で利用可能 <https://github.com/jagregory/abrash-black-book>

¹³以下で利用可能 <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html>

- [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]¹⁴
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)¹⁵

第8.1.6節アセンブリ言語

Richard Blum — Professional Assembly Language.

第8.1.7節Java

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] ¹⁶.

第8.1.8節UNIX

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

第8.1.9節プログラミング一般

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002). 本からのトリックやハックは、分岐命令が高価であるRISC CPUにのみ適していたので、今日は関係ないと言う人もいます。それにもかかわらず、これらはブール代数とそれに近いすべての数学を理解するために非常に役立ちます。

第8.1.10節暗号学

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) Ivh, *Crypto 101*¹⁷
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*¹⁸.

¹⁴以下で利用可能 [http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_\(Issue_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

¹⁵以下で利用可能 [https://yurichev.com/ref/ARM%20Cookbook%20\(1994\)/](https://yurichev.com/ref/ARM%20Cookbook%20(1994)/)

¹⁶以下で利用可能 <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>; <http://docs.oracle.com/javase/specs/jvms/se7/html/>

¹⁷以下で利用可能 <https://www.crypto101.io/>

¹⁸以下で利用可能 <https://crypto.stanford.edu/~dabo/cryptobook/>

Afterword

第8.2節Questions?

恥ずかしがらずに著者へメールしてみよう [my emails](#). この本への新たなコンテンツについての提案がありますか？怖がらずにどんな訂正でも送ってください (文法ミスも含め (私の日本語がとってもひどいのを見てるでしょ？))

The author is working on the book a lot, so the page and listing numbers, etc., are changing very rapidly. Please do not refer to page and listing numbers in your emails to me. There is a much simpler method: make a screenshot of the page, in a graphics editor underline the place where you see the error, and send it to the author. He'll fix it much faster. And if you familiar with git and $\LaTeX$ you can fix the error right in the source code:

<https://beginners.re/src/>.

Do not worry to bother me while writing me about any petty mistakes you found, even if you are not very confident. I'm writing for beginners, after all, so beginners' opinions and comments are crucial for my job.

頭字語

	569
OS オペレーティングシステム	xii
PL プログラミング言語	ix
PRNG 擬似乱数生成器	458
ROM 読み取り専用メモリ	102
ALU 算術論理ユニット	34
LIFO 後入れ先出し	39
MSB 最上位ビット	384
LSB 最下位ビット	
ABI アプリケーション・バイナリー・インタフェース	20
RA リターンアドレス	28
PE Portable Executable	6
LR Link Register	8
IDA Hex-Rays によって開発されたインタラクティブなディスアセンブラ・デバッガ	8
MSVC Microsoft Visual C++	
TLS Thread Local Storage	343
AKA 別名	39
CRT C Runtime library	13
CPU Central Processing Unit	2
FPU Floating-Point Unit	v

	570
CISC Complex Instruction Set Computing	25
RISC Reduced Instruction Set Computing	3
BSS Block Started by Symbol	32
SIMD Single Instruction, Multiple Data	239
DBMS Database Management Systems	ix
ISA Instruction Set Architecture	3
SEH Structured Exception Handling	48
ELF Executable and Linkable Format: Linuxを含め*NIXシステムで広く使用される実行 ファイルフォーマット	100
TIB Thread Information Block	343
NOP No Operation	8
BEQ (PowerPC, ARM) Branch if Equal	119
BNE (PowerPC, ARM) Branch if Not Equal	256
RAM Random-Access Memory	4
GCC GNU Compiler Collection	5
ASCII American Standard Code for Information Interchange	355
ASCIIZ ASCII Zero (ヌル終端文字列)	116
VM Virtual Memory	
GPR General Purpose Registers	
BCD Binary-Coded Decimal	546

	571
GDB GNU Debugger	60
FP Frame Pointer	31
JPE Jump Parity Even (x86命令)	290
STMFD Store Multiple Full Descending (ARM命令)	
LDMFD Load Multiple Full Descending (ARM命令)	
STMED Store Multiple Empty Descending (ARM命令)	39
LDMED Load Multiple Empty Descending (ARM命令)	39
STMFA Store Multiple Full Ascending (ARM命令)	39
LDMFA Load Multiple Full Ascending (ARM命令)	39
STMEA Store Multiple Empty Ascending (ARM命令)	39
LDMEA Load Multiple Empty Ascending (ARM命令)	39
APSR (ARM) Application Program Status Register	315
FPSCR (ARM) Floating-Point Status and Control Register	315
JIT Just-In-Time compilation	560
EOF End of File (ファイル終端)	108
DES Data Encryption Standard	547
MIME Multipurpose Internet Mail Extensions	547
URL Uniform Resource Locator	5

用語集

anti-pattern 一般に、よくないと考えられるやり方. [42](#), [96](#)

callee 呼び出された関数. [42](#), [58](#), [84](#), [109](#), [122](#), [125](#), [128](#), [512](#)

caller 呼び出し元の関数. [7](#), [8](#), [10](#), [13](#), [38](#), [58](#), [109](#), [122](#), [123](#), [127](#), [191](#), [512](#)

endianness バイトオーダー: ?? on page ?? [99](#), [420](#)

GiB ギガバイト: 2^{30} または1024メガバイトまたは1073741824バイト. [20](#)

jump offset JMP命令またはJcc命令のオペコードの一部を次の命令のアドレスに追加する必要があります。これが新しい**PC!**の計算方法です。負となる場合もあります. [117](#), [163](#), [164](#)

leaf function 他の関数から呼び出されない関数. [37](#), [41](#)

link register (RISC) リターンアドレスが保存されるレジスタ。これはleaf functionをスタックを使わずに呼び出すのを可能にする. [41](#)

loop unwinding n 回のイテレーションのループコードをコンパイラが生成する代わりに、ループボディを n 回コピーするコードを生成する。ループに使用する命令を削除するため. [228](#)

NaN 非数: float型の数の特殊なケースで、エラーが通知される. [286](#), [308](#)

register allocator CPUレジスタをローカル変数に割り当てるコンパイラの機構. [247](#), [372](#), [513](#)

reverse engineering 時にはクローンするため、どうやって動いているのかを理解しようとする行為. [iv](#)

stack frame 現在の関数に固有の情報（ローカル変数、関数の引数、**RA**など）を含むスタックの一部. [86](#), [123](#)

stdout standard output. [27](#), [46](#), [191](#)

thunk function 単一の役割だけ持つ小さな関数: 他の関数を呼び出す等. [29](#), [476](#)

tracer シンプルなデバッグツールです。以下で詳細を読むことができます: [6.2.1 on page 559](#). [231-233](#)

Windows NT Windows NT, 2000, XP, Vista, 7, 8, 10. [354](#), [509](#)

word PCよりも昔のコンピュータでは、メモリサイズはバイトではなくワードで測定されることがしばしばでした. [545](#), [547-550](#)

インクリメント 1の加算. [21](#), [25](#), [226](#), [230](#), [249](#), [255](#), [396](#), [399](#), [537](#)

スタックポインタ スタックの場所を示すレジスタ. [14](#), [25](#), [39](#), [45](#), [55](#), [68](#), [70](#), [93](#), [125](#)

デクリメント 1の減算. [24](#), [225](#), [226](#), [249](#), [537](#)

ヒープ OSが提供する大きなメモリの塊のことで、アプリケーションが好きなように分割することができる。malloc()/free() を呼び出して使用する. [39](#), [422](#)

商 除算結果. [267](#), [270](#), [272](#), [273](#), [277](#), [526](#)

実数 小数点以下を含む可能性のある数字。これは C/C++ で *float* と *double* です . [267](#)

整数型 通常の数字ですが、実際の数字はありません。boolと列挙型の変数を渡すために使用できます. [283](#)

積 乗算結果. [123](#), [274](#), [277](#), [495](#), [527](#)

索引

Japanese text placeholder, 233, 317

0x0BADF00D, 96

0xCCCCCCCC, 96

Ada, 133

Alpha AXP, 3

Angry Birds, 316, 317

Apollo Guidance Computer, 259

ARM, 255

Addressing modes, 536

ARM1, 550

armel, 278

armhf, 278

ARMモード, 2

Condition codes, 167

D-レジスタ, 277

DCB, 25

hard float, 278

if-then block, 316

Leaf function, 41

Mode switching, 130, 215

mode switching, 28

Optional operators

ASR, 404

LSL, 328, 362, 404, 539

LSR, 404

ROR, 404

RRX, 404

Pipeline, 213

S-レジスタ, 277

soft float, 278

Thumb-2モード, 2, 215, 316, 317

Thumbモード, 2, 169, 215

レジスタ

APSR, 315

FPSCR, 315

Link Register, 24, 25, 41, 68, 215

scratch registers, 256

Z, 119

命令

ADC, 485

ADD, 27, 132, 167, 234, 389, 404

ADDAL, 167

ADDCC, 213

ADDS, 130, 485

ADR, 24, 167

ADRCc, 167, 168, 200, 201

ADRP/ADD pair, 31, 69, 103, 350, 367, 540

ASR, 408

ASRS, 382

B, 68, 167, 169

Bcc, 120, 121, 183

BCS, 169, 318

BEQ, 119, 201

BGE, 169

BIC, 381, 382, 388, 410

BL, 24, 26, 28, 29, 31, 167, 541

BLcc, 167, 168

BLE, 169

BLS, 169

BLX, 28

BNE, 169

BX, 130, 215

CMP, 119, 120, 167, 201, 213, 404

CSEL, 179, 185, 188, 405

EOR, 388

FCMPE, 319

FCSEL, 319

FMOV, 539

FMRS, 389

IT, 188, 316, 345

LDMccFD, 168

LDMEA, 39

LDMED, 39

LDMFA, 39

- LDMFD, 25, 39, 168
- LDP, 32
- LDR, 71, 102, 328, 348, 536
- LDRB, 444
- LDRB.W, 256
- LDRSB, 255
- LSL, 404, 408
- LSL.W, 404
- LSLS, 328, 389
- LSR, 408
- LSRS, 389
- MADD, 130
- MLA, 130
- MOV, 10, 25, 26, 404
- MOVcc, 183, 188
- MOVK, 539
- MOVT.W, 28
- MOVW, 28
- MUL, 132
- MULS, 130
- MVNS, 256
- ORR, 381
- POP, 24-26, 39, 41
- PUSH, 26, 39, 41
- RET, 32
- RSB, 175, 362, 404
- SBC, 485
- STMEA, 39
- STMED, 39
- STMFA, 39, 72
- STMFD, 24, 39
- STMIA, 70
- STMIB, 72
- STP, 30, 69
- STR, 70, 328
- SUB, 70, 362, 404
- SUBEQ, 257
- SUBS, 485
- SXTB, 444
- SXTW, 367
- TEST, 247
- TST, 374, 404
- VADD, 277
- VDIV, 277
- VLDR, 277
- VMOV, 277, 315
- VMOVGT, 315
- VMRS, 315
- VMUL, 277
- XOR, 175, 389
- ARM64
 - lo12, 69
- AT&T構文, 15, 48
- bash, 135
- binary grep, 558
- Binary Ninja, 558
- BinNavi, 558
- binutils, 462
- Booth's multiplication algorithm, 266
- Callbacks, 467
- Canary, 340
- cdecl, 55
- column-major order, 355
- Compiler intrinsic, 46
- Cray, 495, 550
- CryptoMiniSat, 521
- Cygwin, 560
- C標準ライブラリ
 - alloca(), 45, 345
 - assert(), 352
 - localtime_r(), 432
 - longjmp(), 191
 - malloc(), 423
 - memcmp(), 553
 - memcpy(), 15, 84
 - pow(), 280
 - puts(), 27
 - qsort(), 468
 - rand(), 411
 - scanf(), 84
 - strcmp(), 553
 - strcpy(), 15
 - strlen(), 246, 507
 - strtok, 260
- C言語の要素
 - C99, 137
 - bool, 369
 - variable length arrays, 345
 - const, 12, 103
 - for, 226
 - if, 153, 190
 - return, 13, 109, 136
 - switch, 189, 190, 200
 - while, 246
 - ポインタ, 84, 92, 138, 467, 512
 - 前置インクリメント, 536

-
- 前置デクリメント, 536
 - 後置インクリメント, 536
 - 後置デクリメント, 536
 - Data general Nova, 266
 - DES, 494, 513
 - Donald E. Knuth, 549
 - double, 268
 - dtruss, 559
 - Dynamically loaded libraries, 28
 - ELF, 100
 - fastcall, 19, 44, 83, 371
 - fetchmail, 547
 - float, 268
 - FORTRAN, 29
 - Fortran, 355
 - Function epilogue, 38, 68, 71, 168, 444
 - Function prologue, 14, 38, 41, 70, 340
 - Fused multiply-add, 130
 - GDB, 37, 60, 64, 340, 477, 478, 558
 - GHex, 557
 - Glibc, 477
 - Hex-Rays, 135, 243, 368
 - Hiew, 116, 163, 557
 - Honeywell 6070, 547
 - IDA, 110, 462, 555, 558
 - var_?, 70, 93
 - IEEE 754, 268, 384, 458, 521
 - Inline code, 235, 381
 - Integer overflow, 133
 - Intel
 - 8080, 255
 - 8086, 255, 380
 - 80386, 380
 - 80486, 268
 - FPU, 268
 - Intel 4004, 546
 - Intel C++, 12, 495
 - iPod/iPhone/iPad, 23
 - JAD, 6
 - Java, 548, 555
 - jumptable, 206, 215
 - Keil, 23
 - LAPACK, 29
 - Linker, 102
 - Linux, 372
 - libc.so.6, 371, 476
 - LLDB, 558
 - LLVM, 23
 - long double, 268
 - Loop unwinding, 228
 - Mac OS X, 560
 - minifloat, 539
 - MIPS, 3
 - Branch delay slot, 10
 - Global Pointer, 363
 - Load delay slot, 204
 - O32, 78, 83
 - グローバルポインタ, 32
 - レジスタ
 - FCCR, 320
 - 命令
 - ADD, 133
 - ADDIU, 33, 107, 108
 - ADDU, 133
 - AND, 384
 - BC1F, 321
 - BC1T, 321
 - BEQ, 121, 171
 - BLTZ, 176
 - BNE, 171
 - BNEZ, 217
 - C.LT.D, 321
 - J, 8, 10, 34
 - JAL, 134
 - JALR, 33, 134
 - JR, 204
 - LB, 242
 - LBU, 242
 - LI, 542
 - LUI, 33, 107, 108, 387, 542
 - LW, 33, 94, 108, 204, 543
 - MFHI, 133
 - MFLO, 133
 - MTC1, 464
 - MULT, 133
 - NOR, 259
 - OR, 36
 - ORI, 384, 542
 - SB, 242
 - SLL, 217, 261, 407

-
- SLLV, [407](#)
 - SLT, [171](#)
 - SLTIU, [217](#)
 - SLTU, [171](#), [173](#), [217](#)
 - SRL, [267](#)
 - SUBU, [176](#)
 - SW, [78](#)
 - 疑似命令
 - B, [238](#)
 - BEQZ, [173](#)
 - LA, [36](#)
 - LI, [10](#)
 - MOVE, [106](#)
 - NEGU, [176](#)
 - NOP, [36](#), [106](#)
 - NOT, [259](#)
 - MS-DOS, [18](#), [43](#), [343](#)
 - Non-a-numbers (NaNs), [308](#)
 - objdump, [462](#), [558](#)
 - octet, [547](#)
 - OllyDbg, [56](#), [88](#), [99](#), [123](#), [139](#), [157](#), [207](#), [230](#), [250](#), [271](#), [287](#), [298](#), [325](#), [334](#), [337](#), [356](#), [394](#), [420](#), [442](#), [443](#), [449](#), [453](#), [471](#), [558](#)
 - Oracle RDBMS, [12](#), [494](#)
 - Page (memory), [509](#)
 - PDP-11, [536](#)
 - PowerPC, [3](#), [32](#)
 - puts() instead of printf(), [27](#), [91](#), [135](#), [165](#)
 - Quake III Arena, [467](#)
 - rada.re, [17](#)
 - Radare, [558](#)
 - rafind2, [558](#)
 - RAM, [102](#)
 - Raspberry Pi, [23](#)
 - Register allocation, [513](#)
 - Relocation, [28](#)
 - Reverse Polish notation, [322](#)
 - RISC pipeline, [168](#)
 - ROM, [102](#), [103](#)
 - row-major order, [355](#)
 - RSA, [6](#)
 - Security cookie, [340](#)
 - Shadow space, [126](#), [128](#), [523](#)
 - Signed numbers, [155](#)
 - SIMD, [521](#)
 - SSE, [521](#)
 - SSE2, [521](#)
 - strace, [559](#)
 - syscall, [371](#), [559](#)
 - Sysinternals, [560](#)
 - Thumb-2モード, [28](#)
 - thunk-functions, [29](#)
 - TLS, [343](#)
 - tracer, [231](#), [473](#), [475](#), [558](#)
 - UCS-2, [548](#)
 - UNIX
 - chmod, [5](#)
 - od, [557](#)
 - strings, [557](#)
 - xxd, [557](#)
 - Unrolled loop, [235](#), [345](#)
 - UTF-16, [548](#)
 - win32
 - GetOpenFileName, [259](#)
 - WinDbg, [558](#)
 - Windows
 - KERNEL32.DLL, [369](#)
 - MSVCR80.DLL, [469](#)
 - Structured Exception Handling, [48](#), [556](#)
 - TIB, [343](#)
 - Win32, [369](#)
 - x86
 - AVX, [494](#)
 - MMX, [493](#)
 - SSE, [494](#)
 - SSE2, [494](#)
 - フラグ
 - CF, [44](#)
 - レジスタ
 - AF, [546](#)
 - EAX, [109](#), [134](#)
 - EBP, [86](#), [123](#)
 - ESP, [55](#), [86](#)
 - JMP, [211](#)
 - ZF, [109](#), [370](#)
 - フラグ, [109](#), [157](#)
 - 命令
 - ADC, [483](#)
 - ADD, [12](#), [55](#), [123](#)

ADDSD, 522	JNE, 109, 154
ADDSS, 536	JP, 285
ADRCc, 178	JZ, 119, 191
AND, 14, 370, 375, 392, 409, 452	LEA, 86, 126, 426
BSF, 511	LEAVE, 14
BTC, 386	LOOP, 225, 245
BTR, 386	MAXSD, 532
BTS, 386	MOV, 10, 13, 16
CALL, 40	MOVDQA, 499
CDQ, 493	MOVDQU, 499
CMOVcc, 168, 178, 180, 183, 188	MOVSD, 530
CMP, 109	MOVSDX, 530
COMISD, 532	MOVSS, 536
COMISS, 536	MOVSX, 247, 255, 442, 444
CPUID, 449	MOVSLD, 347
DEC, 249	MOVZX, 248, 423
DIVSD, 522	MULSD, 522
FADDP, 270, 276	NOT, 254, 256
FATRET, 402, 403	OR, 375
FCMOVcc, 311	PADDD, 499
FCOM, 297, 308	PCMPEQB, 510
FCOMP, 284	PLMULHW, 495
FDIV, 270	PLMULLD, 495
FDIVP, 270	PMOVMSKB, 510
FDIVR, 276	POP, 12, 39
FLD, 281, 284	PUSH, 12, 14, 39, 40, 86
FMUL, 270	PXOR, 510
FNSTSW, 284, 309	RET, 7, 10, 13, 340
FSCALE, 465	ROL, 402
FSTP, 281	SAHF, 308
FUCOM, 308	SAR, 408
FUCOMI, 311	SBB, 483
FUCOMPP, 308	SETcc, 171, 248, 309
FWAIT, 268	SHL, 261, 324, 408
IMUL, 123, 366	SHR, 267, 408, 452
INC, 249	SHRD, 491
INT, 43	SUB, 13, 14, 109, 191
JA, 155, 310	TEST, 247, 370, 374, 409
JAE, 155	XOR, 13, 109, 254
JB, 155	x86-64, 18, 19, 63, 84, 91, 118, 125, 512, 521
JBE, 155	Xcode, 23
Jcc, 121, 182	Z80, 546
JE, 191	
JG, 155	インテル構文, 15, 23
JGE, 154	グローバル変数, 96
JL, 155	コンパイラアノマリ, 182, 366, 382, 403
JLE, 154	スタック, 39, 122, 191
JMP, 40	Stack frame, 86
JNBE, 309	

スタックオーバーフロー, [40](#)
バッファオーバーフロー, [332](#), [339](#)
パンチカード, [322](#)

位置独立コード, [24](#)

再帰, [38](#), [40](#)

糖衣構文, [190](#)